

O'REILLY®

Wydanie V

Python

Wprowadzenie



Helion 

Mark Lutz

Tytuł oryginału: Learning Python, 5th Edition

Tłumaczenie: Grzegorz Kowalczyk (Przedmowa, rozdz. 1 – 29),
Andrzej Watrak (rozdz. 30 – 41, dodatki)
z wykorzystaniem fragmentów książki "Python. Wprowadzenie. Wydanie IV" w przekładzie Anny
Trojan i Marka Pętlickiego

ISBN: 978-83-283-6150-8

© 2020 Helion SA

Authorized Polish translation of the English edition of Learning Python, 5th Edition ISBN
9781449355739 © 2013 Mark Lutz

This translation is published and sold by permission of O'Reilly Media, Inc.,
which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any
means, electronic or mechanical, including photocopying, recording or by any information storage
retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej
publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną,
fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje
naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich
właścicieli.

Autor oraz Helion SA dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne
i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym
ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Helion SA nie ponoszą również
żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych
w książce.

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<http://helion.pl/user/opinie/pytho5>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- **Lubię to!** » Nasza społeczność

Spis treści

Przedmowa	33
Część I. Wprowadzenie	51
1. Pytania i odpowiedzi dotyczące Pythona	53
Dlaczego ludzie używają Pythona?	53
Jakość oprogramowania	54
Wydajność programistów	55
Czy Python jest językiem skryptowym?	55
Jakie są wady języka Python?	57
Kto dzisiaj używa Pythona?	59
Co mogę zrobić za pomocą Pythona?	61
Programowanie systemowe	61
Graficzne interfejsy użytkownika (GUI)	62
Skrypty internetowe	62
Integracja komponentów	63
Programowanie bazodanowe	63
Szybkie prototypowanie	64
Programowanie numeryczne i naukowe	64
I dalej: gry, przetwarzanie obrazu, wyszukiwanie danych, robotyka, Excel...	65
Jak Python jest rozwijany i wspierany?	66
Kompromisy związane z modelem open source	66
Jakie są techniczne mocne strony Pythona?	67
Jest zorientowany obiektowo i funkcyjny	67
Jest darmowy	68
Jest przenośny	68
Ma duże możliwości	69
Można go łączyć z innymi językami	70
Jest względnie łatwy w użyciu	71

Jest względnie łatwy do nauczenia się	71
Zawdzięcza swoją nazwę Monty Pythonowi	72
Jak Python wygląda na tle innych języków?	72
Podsumowanie rozdziału	74
Sprawdź swoją wiedzę — quiz	74
Sprawdź swoją wiedzę — odpowiedzi	75
2. Jak Python wykonuje programy?	79
Wprowadzenie do interpretera Pythona	79
Wykonywanie programu	81
Z punktu widzenia programisty	81
Z punktu widzenia Pythona	82
Warianty modeli wykonywania	85
Alternatywne implementacje Pythona	85
Narzędzia do optymalizacji działania programu	89
Zamrożone pliki binarne	91
Przyszłe możliwości?	92
Podsumowanie rozdziału	93
Sprawdź swoją wiedzę — quiz	93
Sprawdź swoją wiedzę — odpowiedzi	93
3. Jak wykonuje się programy?	95
Interaktywny wiersz poleceń	95
Uruchamianie sesji interaktywnej	96
Ścieżka systemowa	98
Nowe opcje systemu Windows w wersji 3.3: PATH, Launcher	98
Gdzie zapisywać programy — katalogi z kodem źródłowym	99
Czego nie wpisywać — znaki zachęty i komentarze	100
Interaktywne wykonywanie kodu	101
Do czego służy sesja interaktywna	103
Uwagi praktyczne — wykorzystywanie sesji interaktywnej	104
Systemowy wiersz poleceń i pliki źródłowe	106
Pierwszy skrypt	107
Wykonywanie plików z poziomu wiersza poleceń powłoki	108
Sposoby użycia wiersza poleceń	109
Uwagi praktyczne — wykorzystywanie wierszy poleceń i plików	110
Skrypty wykonywalne w stylu uniksowym — #!	111
Podstawy skryptów uniksowych	112
Sztuczka z wyszukiwaniem programu przy użyciu polecenia env w systemie Unix	112
Python 3.3 launcher — #! w systemie Windows	113

Klikanie ikon plików	114
Podstawowe zagadnienia związane z klikaniem ikon plików	114
Kliknięcie ikony w systemie Windows	115
Sztuczka z funkcją input	117
Inne ograniczenia programów uruchamianych kliknięciem ikony	119
Importowanie i przeładowywanie modułów	119
Podstawy importowania i przeładowywania modułów	119
Więcej o modułach — atrybuty	121
Uwagi praktyczne — instrukcje import i reload	124
Wykorzystywanie funkcji exec do wykonywania plików modułów	125
Interfejs użytkownika środowiska IDLE	126
Szczegóły uruchamiania środowiska IDLE	127
Podstawy środowiska IDLE	128
Wybrane funkcje środowiska IDLE	130
Zaawansowane narzędzia środowiska IDLE	130
Uwagi praktyczne — korzystanie ze środowiska IDLE	131
Inne środowiska IDE	133
Inne opcje wykonywania kodu	135
Osadzanie wywołań	135
Zamrożone binarne pliki wykonywalne	136
Uruchamianie kodu z poziomu edytora tekstu	136
Jeszcze inne możliwości uruchamiania	136
Przyszłe możliwości?	137
Jaką opcję wybrać?	137
Podsumowanie rozdziału	139
Sprawdź swoją wiedzę — quiz	139
Sprawdź swoją wiedzę — odpowiedzi	140
Sprawdź swoją wiedzę — ćwiczenia do części pierwszej	141

Część II. Typy i operacje 145

4. Wprowadzenie do typów obiektów Pythona 147

Hierarchia pojęć w Pythonie	147
Dlaczego korzystamy z typów wbudowanych	148
Najważniejsze typy danych w Pythonie	149
Liczby	151
Łańcuchy znaków	153
Operacje na sekwencjach	153
Niezmienność	155
Metody specyficzne dla typu	156
Uzyskiwanie pomocy	157

Inne sposoby kodowania łańcuchów znaków	159
Ciągi znaków w formacie Unicode	160
Dopasowywanie wzorców	162
Listy	162
Operacje na typach sekwencyjnych	163
Operacje specyficzne dla typu	163
Sprawdzanie granic	164
Zagnieżdżanie	164
Listy składane	165
Słowniki	167
Operacje na odwzorowaniach	167
Zagnieżdżanie raz jeszcze	169
Brakujące klucze — testowanie za pomocą if	170
Sortowanie kluczy — pętla for	172
Iteracja i optymalizacja	173
Krotki	175
Do czego służą krotki	176
Pliki	176
Pliki binarne	177
Pliki tekstowe Unicode	178
Inne narzędzia podobne do plików	180
Inne typy podstawowe	180
Jak zepsuć elastyczność kodu	182
Klasy definiowane przez użytkownika	183
I wszystko inne	184
Podsumowanie rozdziału	184
Sprawdź swoją wiedzę — quiz	185
Sprawdź swoją wiedzę — odpowiedzi	185
5. Typy liczbowe	187
Podstawy typów liczbowych Pythona	187
Literały liczbowe	188
Wbudowane narzędzia liczbowe	190
Operatory wyrażeń Pythona	190
Liczby w akcji	195
Zmienne i podstawowe wyrażenia	195
Formaty wyświetlania liczb	197
Porównania — zwykłe i łączone	199
Dzielenie — klasyczne, bez reszty i prawdziwe	200
Precyzja liczb całkowitych	204
Liczby zespolone	205

Notacja szesnastkowa, ósemkowa i dwójkowa — literały i konwersje	205
Operacje na poziomie bitów	208
Inne wbudowane narzędzia numeryczne	209
Inne typy liczbowe	211
Typ Decimal (liczby dziesiętne)	211
Typ Fraction (liczby ułamkowe)	214
Zbiory	218
Wartości Boolean	225
Rozszerzenia numeryczne	226
Podsumowanie rozdziału	226
Sprawdź swoją wiedzę — quiz	227
Sprawdź swoją wiedzę — odpowiedzi	227
6. Wprowadzenie do typów dynamicznych	229
Sprawa brakujących deklaracji typu	229
Zmienne, obiekty i referencje	229
Typy powiązane są z obiektami, a nie ze zmiennymi	231
Obiekty są uwalniane	232
Referencje współdzielone	234
Referencje współdzielone a modyfikacje w miejscu	236
Referencje współdzielone a równość	237
Typy dynamiczne są wszędzie	239
Podsumowanie rozdziału	240
Sprawdź swoją wiedzę — quiz	240
Sprawdź swoją wiedzę — odpowiedzi	240
7. Łańcuchy znaków	243
Co znajdziesz w tym rozdziale	243
Unicode — krótka historia	244
Łańcuchy znaków — podstawy	244
Literały łańcuchów znaków	246
Łańcuchy znaków w apostrofach i cudzysłowach są tym samym	247
Sekwencje ucieczki reprezentują znaki specjalne	248
Surowe łańcuchy znaków blokują sekwencje ucieczki	251
Potrojne cudzysłowy i apostrofy kodują łańcuchy znaków będące wielowierszowymi blokami	252
Łańcuchy znaków w akcji	254
Podstawowe operacje	254
Indeksowanie i wycinki	255
Narzędzia do konwersji łańcuchów znaków	259
Modyfikowanie łańcuchów znaków	262

Metody łańcuchów znaków	263
Składnia wywoływania metod	264
Metody typów znakowych	264
Przykłady metod łańcuchów znaków — modyfikowanie	265
Przykłady metod łańcuchów znaków — analiza składniowa tekstu	267
Inne często używane metody łańcuchów znaków	268
Oryginalny moduł string (usunięty w wersji 3.0)	269
Wyrażenia formatujące łańcuchy znaków	270
Formatowanie łańcuchów tekstu	
z użyciem wyrażeń formatujących — podstawy	271
Składnia zaawansowanych wyrażeń formatujących	272
Przykłady zaawansowanych wyrażeń formatujących	274
Wyrażenia formatujące oparte na słowniku	275
Formatowanie łańcuchów z użyciem metody format	276
Podstawy	276
Używanie kluczy, atrybutów i przesunięć	277
Zaawansowana składnia wywołań metody format	278
Przykłady zaawansowanego formatowania łańcuchów znaków	
z użyciem metody format	279
Porównanie metody format z wyrażeniami formatującymi	281
Dlaczego miałbyś korzystać z metody format	284
Generalne kategorie typów	289
Typy z jednej kategorii współdzielą zbiory operacji	289
Typy mutowalne można modyfikować w miejscu	290
Podsumowanie rozdziału	291
Sprawdź swoją wiedzę — quiz	291
Sprawdź swoją wiedzę — odpowiedzi	291
8. Listy oraz słowniki	293
Listy	293
Listy w akcji	295
Podstawowe operacje na listach	295
Iteracje po listach i składanie list	296
Indeksowanie, wycinki i macierze	297
Modyfikacja list w miejscu	298
Słowniki	303
Słowniki w akcji	306
Podstawowe operacje na słownikach	306
Modyfikacja słowników w miejscu	307
Inne metody słowników	308
Przykład — baza danych o filmach	309

Uwagi na temat korzystania ze słowników	311
Inne sposoby tworzenia słowników	315
Zmiany dotyczące słowników w Pythonie 3.x i 2.7	317
Podsumowanie rozdziału	324
Sprawdź swoją wiedzę — quiz	325
Sprawdź swoją wiedzę — odpowiedzi	325
9. Krotki, pliki i wszystko inne	327
Krotki	328
Krotki w akcji	329
Dlaczego istnieją listy i krotki	331
Repetytorium: rekordy — krotki nazwane	332
Pliki	334
Otwieranie plików	335
Wykorzystywanie plików	336
Pliki w akcji	337
Pliki tekstowe i binarne — krótka historia	339
Przechowywanie obiektów Pythona w plikach i przetwarzanie ich	340
Przechowywanie natywnych obiektów Pythona — moduł pickle	342
Przechowywanie obiektów Pythona w formacie JSON	343
Przechowywanie spakowanych danych binarnych — moduł struct	345
Menedżery kontekstu plików	346
Inne narzędzia powiązane z plikami	346
Przegląd i podsumowanie podstawowych typów obiektów	347
Elastyczność obiektów	349
Referencje a kopie	350
Porównania, testy równości i prawda	352
Prawda czy fałsz, czyli znaczenie True i False w Pythonie	355
Hierarchie typów Pythona	357
Obiekty typów	359
Inne typy w Pythonie	360
Pułapki typów wbudowanych	360
Przypisanie tworzy referencje, nie kopie	360
Powtórzenie dodaje jeden poziom zagłębienia	361
Uwaga na cykliczne struktury danych	362
Typów niemutowalnych nie można modyfikować w miejscu	362
Podsumowanie rozdziału	363
Sprawdź swoją wiedzę — quiz	363
Sprawdź swoją wiedzę — odpowiedzi	364
Sprawdź swoją wiedzę — ćwiczenia do części drugiej	364

Część III. Instrukcje i składnia	369
10. Wprowadzenie do instrukcji Pythona	371
Raz jeszcze o hierarchii pojęciowej języka Python	371
Instrukcje Pythona	372
Historia dwóch if	374
Co dodaje Python	374
Co usuwa Python	374
Skąd bierze się składnia z użyciem wcięć	376
Kilka przypadków specjalnych	379
Szybki przykład — interaktywne pętle	381
Prosta pętla interaktywna	381
Wykonywanie obliczeń na danych wpisywanych przez użytkownika	382
Obsługa błędów poprzez sprawdzanie danych wejściowych	383
Obsługa błędów za pomocą instrukcji try	384
Kod zagnieżdżony na trzy poziomy głębokości	386
Podsumowanie rozdziału	387
Sprawdź swoją wiedzę — quiz	387
Sprawdź swoją wiedzę — odpowiedzi	387
11. Przypisania, wyrażenia i wyświetlanie	389
Instrukcje przypisania	389
Formy instrukcji przypisania	390
Przypisanie sekwencji	391
Rozszerzona składnia rozpakowania sekwencji w Pythonie 3.x	394
Przypisanie z wieloma celami	398
Przypisania rozszerzone	399
Reguły dotyczące nazw zmiennych	401
Instrukcje wyrażeń	404
Instrukcje wyrażeń i modyfikacje w miejscu	406
Polecenia print	406
Funkcja print z Pythona 3.x	407
Instrukcja print w Pythonie 2.x	410
Przekierowanie strumienia wyjściowego	411
Wyświetlanie niezależne od wersji	415
Podsumowanie rozdziału	418
Sprawdź swoją wiedzę — quiz	418
Sprawdź swoją wiedzę — odpowiedzi	418

12. Testy if i reguły składni	421
Instrukcje if	421
Ogólny format	421
Proste przykłady	422
Rozgałęzienia kodu	422
Reguły składni Pythona raz jeszcze	424
Ograniczniki bloków — reguły tworzenia wcięć	425
Ograniczniki instrukcji — wiersze i znaki kontynuacji	428
Kilka przypadków specjalnych	428
Testy prawdziwości i testy logiczne	430
Wyrażenie trójargumentowe if/else	432
Podsumowanie rozdziału	435
Sprawdź swoją wiedzę — quiz	435
Sprawdź swoją wiedzę — odpowiedzi	435
13. Pętle while i for	437
Pętle while	437
Ogólny format	438
Przykłady	438
Instrukcje break, continue, pass oraz else w pętli	439
Ogólny format pętli	439
Instrukcja pass	440
Instrukcja continue	441
Instrukcja break	441
Klauzula else pętli	442
Pętle for	444
Ogólny format	445
Przykłady	445
Techniki tworzenia pętli	451
Pętle z licznikami — range	452
Skanowanie sekwencji — pętla while z funkcją range kontra pętla for	453
Przetrasowania sekwencji — funkcje range i len	454
Przechodzenie niewyczerpujące — range kontra wycinki	454
Modyfikowanie list — range kontra listy składane	455
Przechodzenie równoległe — zip oraz map	456
Generowanie wartości przesunięcia i elementów — enumerate	459
Podsumowanie rozdziału	463
Sprawdź swoją wiedzę — quiz	463
Sprawdź swoją wiedzę — odpowiedzi	463

14. Iteracje i listy składane	465
Iteracje — pierwsze spojrzenie	465
Protokół iteracyjny — iteratory plików	466
Iterowanie ręczne — iter i next	469
Inne wbudowane typy iterowalne	472
Listy składane — wprowadzenie	474
Podstawy list składanych	474
Wykorzystywanie list składanych w plikach	475
Rozszerzona składnia list składanych	476
Inne konteksty iteracyjne	478
Nowe obiekty iterowalne w Pythonie 3.x	483
Wpływ na kod w wersji 2.x — zalety i wady	483
Obiekt iterowalny range	484
Obiekty iterowalne map, zip i filter	485
Iteratory wielokrotne kontra pojedyncze	486
Obiekty iterowalne — widoki słownika	487
Inne zagadnienia związane z iteracjami	489
Podsumowanie rozdziału	489
Sprawdź swoją wiedzę — quiz	490
Sprawdź swoją wiedzę — odpowiedzi	490
15. Wprowadzenie do dokumentacji	491
Źródła dokumentacji Pythona	491
Komentarze ze znakami #	492
Funkcja dir	492
Notki dokumentacyjne — <code>__doc__</code>	494
PyDoc — funkcja help	497
PyDoc — raporty HTML	500
Nie tylko notki docstrings — pakiet Sphinx	508
Zbiór standardowej dokumentacji	509
Zasoby internetowe	509
Publikowane książki	510
Często spotykane problemy programistyczne	511
Podsumowanie rozdziału	513
Sprawdź swoją wiedzę — quiz	513
Sprawdź swoją wiedzę — odpowiedzi	513
Ćwiczenia do części trzeciej	514

Część IV . Funkcje i generatory	517
16. Podstawy funkcji	519
Dlaczego używamy funkcji	519
Tworzenie funkcji	521
Instrukcje def	522
Instrukcja def uruchamiana jest w czasie wykonania	523
Pierwszy przykład — definicje i wywoływanie	524
Definicja	524
Wywołanie	524
Polimorfizm w Pythonie	525
Drugi przykład — przecinające się sekwencje	526
Definicja	526
Wywołania	527
Raz jeszcze o polimorfizmie	528
Zmienne lokalne	528
Podsumowanie rozdziału	529
Sprawdź swoją wiedzę — quiz	529
Sprawdź swoją wiedzę — odpowiedzi	529
17. Zasięgi	531
Podstawy zasięgów w Pythonie	531
Reguły dotyczące zasięgów	532
Rozwiązywanie nazw — reguła LEGB	534
Przykład zasięgu	536
Zasięg wbudowany	537
Instrukcja global	540
Projektowanie programów:	
minimalizowanie stosowania zmiennych globalnych	541
Projektowanie programów:	
minimalizowanie modyfikacji dokonywanych pomiędzy plikami	543
Inne metody dostępu do zmiennych globalnych	544
Zasięgi a funkcje zagnieżdżone	545
Szczegóły dotyczące zasięgów zagnieżdżonych	546
Przykłady zasięgów zagnieżdżonych	546
Funkcje fabrykujące: domknięcia	547
Zachowywanie stanu zasięgu zawierającego	
za pomocą argumentów domyślnych	550
Instrukcja nonlocal w Pythonie 3.x	553
Podstawy instrukcji nonlocal	554
Instrukcja nonlocal w akcji	555

Czemu służą zmienne nonlocal? Opcje zachowania stanu	558
Zachowanie stanu: zmienne nonlocal (tylko w wersji 3.x)	558
Zachowanie stanu: zmienne globalne — tylko jedna kopia	559
Zachowanie stanu: klasy — jawne atrybuty (wprowadzenie)	559
Zachowanie stanu: atrybuty funkcji (w wersjach 3.x i 2.x)	561
Podsumowanie rozdziału	564
Sprawdź swoją wiedzę — quiz	565
Sprawdź swoją wiedzę — odpowiedzi	566
18. Argumenty	567
Podstawy przekazywania argumentów	567
Argumenty a współdzielone referencje	568
Unikanie modyfikacji argumentów mutowalnych	570
Symulowanie parametrów wyjścia i wielu wyników działania	571
Specjalne tryby dopasowywania argumentów	572
Podstawy dopasowywania argumentów	572
Składnia dopasowania argumentów	573
Dopasowywanie argumentów — szczegóły	575
Przykłady ze słowami kluczowymi i wartościami domyślnymi	576
Przykłady dowolnych argumentów	578
Argumenty tylko ze słowami kluczowymi (z Pythona 3.x)	582
Przykład z funkcją obliczającą minimum	585
Pełne rozwiązanie	586
Dodatkowy bonus	587
Puenta	588
Uogólnione funkcje działające na zbiorach	588
Emulacja funkcji print z Pythona 3.0	590
Wykorzystywanie argumentów ze słowami kluczowymi	592
Podsumowanie rozdziału	594
Sprawdź swoją wiedzę — quiz	594
Sprawdź swoją wiedzę — odpowiedzi	595
19. Zaawansowane zagadnienia dotyczące funkcji	597
Koncepcje projektowania funkcji	597
Funkcje rekurencyjne	599
Sumowanie z użyciem rekurencji	600
Implementacje alternatywne	600
Pętle a rekurencja	601
Obsługa dowolnych struktur	602
Obiekty funkcji — atrybuty i adnotacje	606
Pośrednie wywołania funkcji — obiekty „pierwszej klasy”	606
Introspekcja funkcji	607

Atrybuty funkcji	608
Adnotacje funkcji w Pythonie 3.x	609
Funkcje anonimowe — lambda	611
Podstawy wyrażeń lambda	611
Po co używamy wyrażeń lambda	612
Jak (nie) zaciemniać kodu napisanego w Pythonie	614
Zasięgi: wyrażenia lambda również można zagnieżdżać	615
Narzędzia programowania funkcyjnego	617
Odwzorowywanie funkcji na obiekty iterowalne — map	617
Wybieranie elementów obiektów iterowalnych — funkcja filter	619
Łączenie elementów obiektów iterowalnych — funkcja reduce	619
Podsumowanie rozdziału	620
Sprawdź swoją wiedzę — quiz	621
Sprawdź swoją wiedzę — odpowiedzi	621
20. Listy składane i generatory	623
Listy składane i narzędzia funkcyjne	623
Listy składane kontra funkcja map	624
Dodajemy warunki i pętle zagnieżdżone — filter	625
Przykład — listy składane i macierze	627
Nie nadużywaj list składanych: reguła KISS	630
Funkcje i wyrażenia generatorów	631
Funkcje generatorów — yield kontra return	633
Wyrażenia generatorów — obiekty iterowalne spotykają złożenia	638
Funkcje generatorów a wyrażenia generatorów	643
Generatory są obiektami o jednorazowej iteracji	644
Generowanie wyników we wbudowanych typach, narzędziach i klasach	646
Przykład — generowanie mieszanych sekwencji	649
Nie nadużywaj generatorów: reguła EIBTI	654
Przykład — emulowanie funkcji zip i map za pomocą narzędzi iteracyjnych	656
Podsumowanie obiektów składanych	662
Zakresy i zmienne składane	662
Zrozumieć zbiory i słowniki składane	663
Rozszerzona składnia zbiorów i słowników składanych	664
Podsumowanie rozdziału	665
Sprawdź swoją wiedzę — quiz	665
Sprawdź swoją wiedzę — odpowiedzi	666
21. Wprowadzenie do pomiarów wydajności	667
Pomiary wydajności iteracji	667
Moduł pomiaru czasu domowej roboty	668
Skrypt mierzący wydajność	672

Wyniki pomiarów czasu	673
Inne rozwiązania dla modułu do pomiaru czasu	676
Inne sugestie	679
Mierzenie czasu iteracji z wykorzystaniem modułu <code>timeit</code>	680
Podstawowe reguły korzystania z modułu <code>timeit</code>	680
Moduł i skrypt testujący z użyciem modułu <code>timeit</code>	685
Wyniki działania skryptu testującego	686
Jeszcze trochę zabawy z mierzeniem wydajności	689
Inne zagadnienia związane z mierzeniem szybkości działania kodu — test <code>pystone</code>	693
Pułapki związane z funkcjami	694
Lokalne nazwy są wykrywane w sposób statyczny	694
Wartości domyślne i obiekty mutowalne	695
Funkcje, które nie zwracają wyników	697
Różne problemy związane z funkcjami	698
Podsumowanie rozdziału	698
Sprawdź swoją wiedzę — quiz	699
Sprawdź swoją wiedzę — odpowiedzi	699
Sprawdź swoją wiedzę — ćwiczenia do części czwartej	700

Część V. Moduły i pakiety 703

22. Moduły — wprowadzenie	705
Dlaczego używamy modułów	705
Architektura programu w Pythonie	706
Struktura programu	707
Importowanie i atrybuty	707
Moduły biblioteki standardowej	709
Jak działa importowanie	710
1. Odszukanie modułu	710
2. Kompilowanie (o ile jest to potrzebne)	711
3. Wykonanie	712
Pliki kodu bajtowego — <code>__pycache__</code> w Pythonie 3.2+	712
Modele plików kodu bajtowego w akcji	713
Ścieżka wyszukiwania modułów	714
Konfigurowanie ścieżki wyszukiwania	717
Wariacje ścieżki wyszukiwania modułów	717
Lista <code>sys.path</code>	717
Wybór pliku modułu	718
Podsumowanie rozdziału	721
Sprawdź swoją wiedzę — quiz	721
Sprawdź swoją wiedzę — odpowiedzi	722

23. Podstawy tworzenia modułów	723
Tworzenie modułów	723
Nazwy modułów	723
Inne rodzaje modułów	724
Używanie modułów	724
Instrukcja import	725
Instrukcja from	725
Instrukcja from *	725
Operacja importowania jest przeprowadzana tylko raz	726
Instrukcje import oraz from są przypisaniami	727
Równoważność instrukcji import oraz from	728
Potencjalne pułapki związane z użyciem instrukcji from	729
Przestrzenie nazw modułów	730
Pliki generują przestrzenie nazw	731
Słowniki przestrzeni nazw: <code>__dict__</code>	732
Kwalifikowanie nazw atrybutów	733
Importowanie a zasięgi	734
Zagnieżdżanie przestrzeni nazw	734
Przeładowywanie modułów	735
Podstawy przeładowywania modułów	736
Przykład przeładowywania z użyciem reload	737
Podsumowanie rozdziału	739
Sprawdź swoją wiedzę — quiz	739
Sprawdź swoją wiedzę — odpowiedzi	740
24. Pakiety modułów	741
Podstawy importowania pakietów	741
Pakiety a ustawienia ścieżki wyszukiwania	742
Pliki pakietów <code>__init__.py</code>	743
Przykład importowania pakietu	745
Instrukcja from a instrukcja import w importowaniu pakietów	746
Do czego służy importowanie pakietów	747
Historia trzech systemów	748
Względne importowanie pakietów	751
Zmiany w Pythonie 3.0	751
Podstawy importowania względnego	752
Do czego służą importy względne	753
Zasięg importów względnych	755
Podsumowanie reguł wyszukiwania modułów	756
Importy względne w działaniu	757
Pułapki związane z importem względnym w pakietach: zastosowania mieszane	762

Pakiety przestrzeni nazw w Pythonie 3.3	767
Semantyka pakietów przestrzeni nazw	768
Wpływ na zwykłe pakiety: opcjonalne pliki <code>__init__.py</code>	769
Pakiety przestrzeni nazw w akcji	770
Zagnieżdżanie pakietów przestrzeni nazw	771
Pliki nadal mają pierwszeństwo przed katalogami	772
Podsumowanie rozdziału	774
Sprawdź swoją wiedzę — quiz	775
Sprawdź swoją wiedzę — odpowiedzi	775
25. Zaawansowane zagadnienia związane z modułami	777
Koncepcje związane z projektowaniem modułów	777
Ukrywanie danych w modułach	779
Minimalizacja niebezpieczeństw użycia <code>from * — _X</code> oraz <code>__all__</code>	779
Włączanie opcji z przyszłych wersji Pythona: <code>__future__</code>	780
Mieszane tryby użycia — <code>__name__</code> oraz <code>__main__</code>	781
Testy jednostkowe z wykorzystaniem atrybutu <code>__name__</code>	782
Przykład — kod działający w dwóch trybach	783
Symbole walut: Unicode w akcji	786
Notki dokumentacyjne: dokumentacja modułu w działaniu	787
Modyfikacja ścieżki wyszukiwania modułów	789
Rozszerzenie <code>as</code> dla instrukcji <code>import</code> oraz <code>from</code>	790
Przykład — moduły są obiektami	791
Importowanie modułów z użyciem nazwy w postaci ciągu znaków	793
Uruchamianie ciągów znaków zawierających kod	793
Bezpośrednie wywołania: dwie opcje	794
Przykład — przechodnie przeładowywanie modułów	795
Przeładowywanie rekurencyjne	795
Rozwiązania alternatywne	798
Pułapki związane z modułami	802
Kolizje nazw modułów: pakiety i importowanie względne w pakietach	802
W kodzie najwyższego poziomu kolejność instrukcji ma znaczenie	803
Instrukcja <code>from</code> kopiuje nazwy, jednak łączy już nie	803
Instrukcja <code>from *</code> może zaciemnić znaczenie zmiennych	804
Funkcja <code>reload</code> może nie mieć wpływu na obiekty importowane za pomocą <code>from</code>	805
Funkcja <code>reload</code> i instrukcja <code>from</code> a testowanie interaktywne	805
Rekurencyjne importowanie za pomocą <code>from</code> może nie działać	806
Podsumowanie rozdziału	808
Sprawdź swoją wiedzę — quiz	808
Sprawdź swoją wiedzę — odpowiedzi	808
Sprawdź swoją wiedzę — ćwiczenia do części piątej	809

Część VI. Klasy i programowanie zorientowane obiektowo813

26. Programowanie zorientowane obiektowo — wprowadzenie	815
Po co używa się klas	816
Programowanie zorientowane obiektowo z dystansu	817
Wyszukiwanie atrybutów dziedziczonych	817
Klasy a instancje	820
Wywołania metod klasy	820
Tworzenie drzew klas	821
Przeciążanie operatorów	823
Programowanie zorientowane obiektowo oparte jest na ponownym wykorzystaniu kodu	824
Podsumowanie rozdziału	827
Sprawdź swoją wiedzę — quiz	827
Sprawdź swoją wiedzę — odpowiedzi	828
27. Podstawy tworzenia klas	829
Klasy generują wiele obiektów instancji	829
Obiekty klas udostępniają zachowania domyślne	830
Obiekty instancji są rzeczywistymi elementami	830
Pierwszy przykład	831
Klasy dostosowujemy do własnych potrzeb przez dziedziczenie	833
Drugi przykład	834
Klasy są atrybutami w modułach	836
Klasy mogą przechwytywać operatory Pythona	837
Trzeci przykład	838
Po co przeciążamy operatory	840
Najprostsza klasa Pythona na świecie	841
Jeszcze kilka słów o rekordach: klasy kontra słowniki	844
Podsumowanie rozdziału	846
Sprawdź swoją wiedzę — quiz	846
Sprawdź swoją wiedzę — odpowiedzi	847
28. Bardziej realistyczny przykład	849
Krok 1. — tworzenie instancji	850
Tworzenie konstruktorów	850
Testowanie w miarę pracy	851
Wykorzystywanie kodu na dwa sposoby	853
Krok 2. — dodawanie metod	854
Tworzenie kodu metod	856
Krok 3. — przeciążanie operatorów	858
Udostępnienie sposobów wyświetlania	858

Krok 4. — dostosowywanie zachowania za pomocą klas podrzędnych	860
Tworzenie klas podrzędnych	861
Rozszerzanie metod — niepoprawny sposób	861
Rozszerzanie metod — poprawny sposób	862
Polimorfizm w akcji	863
Dziedziczenie, dostosowanie do własnych potrzeb i rozszerzenie	865
Programowanie zorientowane obiektowo — idea	866
Krok 5. — dostosowanie do własnych potrzeb także konstruktorów	866
Programowanie zorientowane obiektowo jest prostsze, niż się wydaje	868
Inne sposoby łączenia klas	869
Krok 6. — wykorzystywanie narzędzi do introspekcji	872
Specjalne atrybuty klas	873
Uniwersalne narzędzie do wyświetlania	874
Atrybuty instancji a atrybuty klas	875
Nazwy w klasach narzędziowych	876
Ostateczna postać naszych klas	877
Krok 7. i ostatni — przechowywanie obiektów w bazie danych	879
Obiekty pickle i shelve	879
Przechowywanie obiektów w bazie danych za pomocą shelve	880
Interaktywna eksploracja obiektów shelve	882
Uaktualnianie obiektów w pliku shelve	884
Przyszłe kierunki rozwoju	885
Podsumowanie rozdziału	887
Sprawdź swoją wiedzę — quiz	887
Sprawdź swoją wiedzę — odpowiedzi	888
29. Szczegóły kodowania klas	891
Instrukcja class	891
Ogólna forma	892
Przykład	892
Metody	894
Przykład metody	895
Wywoływanie konstruktorów klas nadrzędnych	896
Inne możliwości wywoływania metod	896
Dziedziczenie	897
Tworzenie drzewa atrybutów	897
Specjalizacja odziedziczonych metod	898
Techniki interfejsów klas	899
Abstrakcyjne klasy nadrzędne	900
Przestrzenie nazw — cała historia	903
Proste nazwy — globalne, o ile nie są przypisane	903
Nazwy atrybutów — przestrzenie nazw obiektów	903

Zen przestrzeni nazw Pythona — przypisania klasyfikują zmienne	904
Klasy zagnieżdżone — jeszcze kilka słów o regule LEGB	906
Słowniki przestrzeni nazw — przegląd	908
Łączy przestrzeni nazw — przechodzenie w górę drzewa klas	910
Raz jeszcze o notkach dokumentacyjnych	912
Klasy a moduły	914
Podsumowanie rozdziału	914
Sprawdź swoją wiedzę — quiz	915
Sprawdź swoją wiedzę — odpowiedzi	915
30. Przeciążanie operatorów	917
Podstawy	917
Konstruktory i wyrażenia — <code>__init__</code> i <code>__sub__</code>	918
Często spotykane metody przeciążania operatorów	918
Indeksowanie i wycinanie — <code>__getitem__</code> i <code>__setitem__</code>	920
Wycinki	921
Wycinanie i indeksowanie w Pythonie 2.x	923
Metoda <code>__index__</code> w wersji 3.x nie służy do indeksowania!	923
Iteracja po indeksie — <code>__getitem__</code>	924
Obiekty iteratorów — <code>__iter__</code> i <code>__next__</code>	925
Iteratory zdefiniowane przez użytkownika	925
Wiele iteracji po jednym obiekcie	928
Alternatywa: metoda <code>__iter__</code> i instrukcja <code>yield</code>	930
Test przynależności — <code>__contains__</code> , <code>__iter__</code> i <code>__getitem__</code>	935
Dostęp do atrybutów — <code>__getattr__</code> oraz <code>__setattr__</code>	937
Odwołania do atrybutów	937
Przypisywanie wartości i usuwanie atrybutów	938
Inne narzędzia do zarządzania atrybutami	940
Emulowanie prywatności w atrybutach instancji	940
Reprezentacje łańcuchów — <code>__repr__</code> oraz <code>__str__</code>	941
Po co nam dwie metody wyświetlania?	942
Uwagi dotyczące wyświetlania	944
Dodawanie prawostronne i miejscowa modyfikacja: metody <code>__radd__</code> i <code>__iadd__</code>	945
Dodawanie prawostronne	945
Dodawanie w miejscu	948
Wywołania — <code>__call__</code>	949
Interfejsy funkcji i kod oparty na wywołaniach zwrotnych	950
Porównania — <code>__lt__</code> , <code>__gt__</code> i inne	952
Metoda <code>__cmp__</code> w 2.x	953
Testy logiczne — <code>__bool__</code> i <code>__len__</code>	954
Metody logiczne w Pythonie 2.x	955

Destrukcja obiektu — <code>__del__</code>	956
Uwagi dotyczące stosowania destruktorów	957
Podsumowanie rozdziału	958
Sprawdź swoją wiedzę — quiz	958
Sprawdź swoją wiedzę — odpowiedzi	958
31. Projektowanie z użyciem klas	961
Python a programowanie zorientowane obiektowo	961
Polimorfizm to interfejsy, a nie sygnatury wywołań	962
Programowanie zorientowane obiektowo i dziedziczenie — związek „jest”	963
Programowanie zorientowane obiektowo i kompozycja — związki typu „ma”	964
Raz jeszcze procesor strumienia danych	966
Programowanie zorientowane obiektowo a delegacja — obiekty „opakowujące”	969
Pseudoprywatne atrybuty klas	971
Przegląd zniekształcania nazw zmiennych	972
Po co używa się atrybutów pseudoprywatnych	973
Metody są obiektami — z wiązaniem i bez wiązania	975
W wersji 3.x metody niezwiązane są funkcjami	977
Metody związane i inne obiekty wywoływane	978
Klasy są obiektami — uniwersalne fabryki obiektów	981
Do czego służą fabryki	982
Dziedziczenie wielokrotne — klasy mieszane	983
Tworzenie klas mieszanych	984
Inne zagadnienia związane z projektowaniem	1002
Podsumowanie rozdziału	1003
Sprawdź swoją wiedzę — quiz	1003
Sprawdź swoją wiedzę — odpowiedzi	1003
32. Zaawansowane zagadnienia związane z klasami	1005
Rozszerzanie typów wbudowanych	1006
Rozszerzanie typów za pomocą osadzania	1006
Rozszerzanie typów za pomocą klas podrzędnych	1007
Klasy w nowym stylu	1009
Jak nowy jest nowy styl	1010
Nowości w klasach w nowym stylu	1011
Pomijanie instancji we wbudowanych operacjach przy pobieraniu atrybutów	1012
Zmiany w modelu typów	1017
Wszystkie obiekty dziedziczą po klasie <code>object</code>	1020
Zmiany w dziedziczeniu diamentowym	1022
Więcej o kolejności odwzorowywania nazw	1026
Przykład — wiązanie atrybutów ze źródłami dziedziczenia	1028

Nowości w klasach w nowym stylu	1034
Sloty: deklaracje atrybutów	1034
Właściwości klas: dostęp do atrybutów	1043
Narzędzia atrybutów: <code>__getattr__</code> i deskryptory	1045
Inne zmiany i rozszerzenia klas	1046
Metody statyczne oraz metody klasy	1047
Do czego potrzebujemy metod specjalnych	1047
Metody statyczne w 2.x i 3.x	1048
Alternatywy dla metod statycznych	1049
Używanie metod statycznych i metod klas	1051
Zliczanie instancji z użyciem metod statycznych	1052
Zliczanie instancji z metodami klas	1053
Dekoratory i metaklasy — część 1.	1056
Podstawowe informacje o dekoratorach funkcji	1056
Pierwsze spojrzenie na funkcję dekoratora zdefiniowaną przez użytkownika	1058
Pierwsze spojrzenie na dekoratory klas i metaklasy	1059
Dalsza lektura	1061
Wbudowana funkcja <code>super</code> : zmiana na lepsze czy na gorsze?	1062
Wielka debata o funkcji <code>super</code>	1062
Tradycyjny, uniwersalny i ogólny sposób wywoływania klasy nadrzędnej	1063
Podstawy i kompromisy użycia funkcji <code>super</code>	1064
Zalety funkcji <code>super</code> : zmiany drzewa i kierowania metod	1069
Zmiana klasy w trakcie działania programu a funkcja <code>super</code>	1070
Kooperatywne kierowanie metod w drzewie wielokrotnego dziedziczenia	1071
Podsumowanie funkcji <code>super</code>	1082
Pułapki związane z klasami	1083
Modyfikacja atrybutów klas może mieć efekty uboczne	1084
Modyfikowanie mutowalnych atrybutów klas również może mieć efekty uboczne	1085
Dziedziczenie wielokrotne — kolejność ma znaczenie	1086
Zakresy w metodach i klasach	1087
Różne pułapki związane z klasami	1088
Przesadne opakowywanie	1089
Podsumowanie rozdziału	1089
Sprawdź swoją wiedzę — quiz	1090
Sprawdź swoją wiedzę — odpowiedzi	1090
Sprawdź swoją wiedzę — ćwiczenia do części szóstej	1091

Część VII. Wyjątki oraz narzędzia1099

33. Podstawy wyjątków	1101
Po co używa się wyjątków	1101
Role wyjątków	1102
Wyjątki w skrócie	1103
Domyślny program obsługi wyjątków	1103
Przechwytywanie wyjątków	1104
Zgłaszanie wyjątków	1105
Wyjątki zdefiniowane przez użytkownika	1106
Działania końcowe	1106
Podsumowanie rozdziału	1109
Sprawdź swoją wiedzę — quiz	1109
Sprawdź swoją wiedzę — odpowiedzi	1110
34. Szczegółowe informacje dotyczące wyjątków	1111
Instrukcja try/except/else	1111
Jak działa instrukcja try	1112
Części instrukcji try	1113
Część try/else	1115
Przykład — zachowanie domyślne	1116
Przykład — przechwytywanie wbudowanych wyjątków	1117
Instrukcja try/finally	1118
Przykład — działania kończące kod z użyciem try/finally	1119
Połączona instrukcja try/except/finally	1120
Składnia połączonej instrukcji try	1121
Łączenie finally oraz except za pomocą zagnieżdżenia	1121
Przykład połączonego try	1122
Instrukcja raise	1123
Zgłaszanie wyjątków	1124
Zakresy widoczności zmiennych w instrukcjach try i except	1125
Przekazywanie wyjątków za pomocą raise	1126
Łańcuchy wyjątków w Pythonie 3.x — raise from	1127
Instrukcja assert	1128
Przykład — wyłapywanie ograniczeń (ale nie błędów!)	1129
Menedżery kontekstu with/as	1130
Podstawowe zastosowanie	1130
Protokół zarządzania kontekstem	1132
Kilka menedżerów kontekstu w wersjach 3.1, 2.7 i nowszych	1133
Podsumowanie rozdziału	1135
Sprawdź swoją wiedzę — quiz	1135
Sprawdź swoją wiedzę — odpowiedzi	1136

35. Obiekty wyjątków	1137
Wyjątki — powrót do przyszłości	1138
Wyjątki oparte na łańcuchach znaków znikają	1138
Wyjątki oparte na klasach	1139
Tworzenie klas wyjątków	1140
Do czego służą hierarchie wyjątków	1142
Wbudowane klasy wyjątków	1144
Kategorie wbudowanych wyjątków	1146
Domyślne wyświetlanie oraz stan	1147
Własne sposoby wyświetlania	1148
Własne dane oraz zachowania	1149
Udostępnianie szczegółów wyjątku	1150
Udostępnianie metod wyjątków	1150
Podsumowanie rozdziału	1152
Sprawdź swoją wiedzę — quiz	1152
Sprawdź swoją wiedzę — odpowiedzi	1152
36. Projektowanie z wykorzystaniem wyjątków	1155
Zagnieżdżanie programów obsługi wyjątków	1155
Przykład — zagnieżdżanie przebiegu sterowania	1157
Przykład — zagnieżdżanie składniowe	1157
Zastosowanie wyjątków	1159
Wychodzenie z głęboko zagnieżdżonych pętli: instrukcja go to	1159
Wyjątki nie zawsze są błędami	1160
Funkcje mogą sygnalizować warunki za pomocą raise	1160
Zamykanie plików oraz połączeń z serwerem	1161
Debugowanie z wykorzystaniem zewnętrznych instrukcji try	1162
Testowanie kodu wewnątrz tego samego procesu	1163
Więcej informacji na temat funkcji sys.exc_info	1164
Wyświetlanie błędów i śladów stosu	1164
Wskazówki i pułapki dotyczące projektowania wyjątków	1165
Co powinniśmy opakować w try	1166
Jak nie przechwytywać zbyt wiele — unikanie pustych except i wyjątków	1166
Jak nie przechwytywać zbyt mało — korzystanie z kategorii opartych na klasach	1168
Podsumowanie podstaw języka Python	1169
Zbiór narzędzi Pythona	1169
Narzędzia programistyczne przeznaczone do większych projektów	1170
Podsumowanie rozdziału	1174
Sprawdź swoją wiedzę — quiz	1174
Sprawdź swoją wiedzę — odpowiedzi	1175
Sprawdź swoją wiedzę — ćwiczenia do części siódmej	1175

Część VIII. Zagadnienia zaawansowane 1177

37. Łańcuchy znaków Unicode oraz łańcuchy bajtowe 1179

Zmiany w łańcuchach znaków w Pythonie 3.x	1180
Podstawy łańcuchów znaków	1181
Kodowanie znaków	1181
Jak Python zapisuje ciągi znaków w pamięci	1183
Typy łańcuchów znaków Pythona	1185
Pliki binarne i tekstowe	1186
Podstawy kodowania ciągów znaków	1188
Literały tekstowe w Pythonie 3.x	1188
Literały tekstowe w Pythonie 2.x	1190
Konwersje typów ciągów	1190
Kod łańcuchów znaków Unicode	1192
Kod tekstu z zakresu ASCII	1192
Kod tekstu spoza zakresu ASCII	1193
Kodowanie i dekodowanie tekstu spoza zakresu ASCII	1194
Inne techniki kodowania łańcuchów Unicode	1194
Literały bajtowe	1196
Konwersja kodowania	1197
Łańcuchy znaków Unicode w Pythonie 2.x	1198
Deklaracje typu kodowania znaków pliku źródłowego	1200
Wykorzystywanie obiektów bytes z Pythona 3.x	1201
Wywołania metod	1202
Operacje na sekwencjach	1203
Inne sposoby tworzenia obiektów bytes	1203
Mieszanie typów łańcuchów znaków	1204
Obiekt bytearray w wersji 3.x (oraz 2.6 lub nowszej)	1205
Typ bytearray w akcji	1205
Podsumowanie typów ciągów znaków w Pythonie 3.x	1207
Wykorzystywanie plików tekstowych i binarnych	1208
Podstawy plików tekstowych	1208
Tryby tekstowy i binarny w Pythonie 2.x i 3.x	1209
Brak dopasowania typu i zawartości w Pythonie 3.x	1210
Wykorzystywanie plików Unicode	1212
Odczyt i zapis Unicode w Pythonie 3.x	1212
Obsługa BOM w Pythonie 3.x	1213
Pliki Unicode w Pythonie 2.x	1216
Unicode w nazwach plików i w strumieniach	1217

Inne zmiany w narzędziach do przetwarzania łańcuchów znaków w Pythonie 3.x	1218
Moduł dopasowywania wzorców re	1218
Moduł danych binarnych struct	1219
Moduł serializacji obiektów pickle	1221
Narzędzia do analizy składniowej XML	1223
Podsumowanie rozdziału	1227
Sprawdź swoją wiedzę — quiz	1227
Sprawdź swoją wiedzę — odpowiedzi	1228
38. Zarządzane atrybuty	1231
Po co zarządza się atrybutami	1231
Wstawianie kodu wykonywanego w momencie dostępu do atrybutów	1232
Właściwości	1233
Podstawy	1233
Pierwszy przykład	1234
Obliczanie atrybutów	1235
Zapisywanie właściwości w kodzie za pomocą dekoratorów	1236
Deskryptory	1237
Podstawy	1238
Pierwszy przykład	1241
Obliczone atrybuty	1242
Wykorzystywanie informacji o stanie w deskryptorach	1243
Powiązania pomiędzy właściwościami a deskryptorami	1246
Metody <code>__getattr__</code> oraz <code>__getattribute__</code>	1248
Podstawy	1249
Pierwszy przykład	1252
Obliczanie atrybutów	1253
Porównanie metod <code>__getattr__</code> oraz <code>__getattribute__</code>	1255
Porównanie technik zarządzania atrybutami	1256
Przechwytywanie atrybutów wbudowanych operacji	1258
Przykład — sprawdzanie poprawności atrybutów	1266
Wykorzystywanie właściwości do sprawdzania poprawności	1266
Wykorzystywanie deskryptorów do sprawdzania poprawności	1268
Wykorzystywanie metody <code>__getattr__</code> do sprawdzania poprawności	1272
Wykorzystywanie metody <code>__getattribute__</code> do sprawdzania poprawności	1273
Podsumowanie rozdziału	1275
Sprawdź swoją wiedzę — quiz	1275
Sprawdź swoją wiedzę — odpowiedzi	1275

39. Dekoratory	1277
Czym jest dekorator	1277
Zarządzanie wywołaniami oraz instancjami	1278
Zarządzanie funkcjami oraz klasami	1278
Wykorzystywanie i definiowanie dekoratorów	1279
Do czego służą dekoratory	1279
Podstawy	1281
Dekoratory funkcji	1281
Dekoratory klas	1284
Zagnieżdżanie dekoratorów	1287
Argumenty dekoratorów	1288
Dekoratory zarządzają także funkcjami oraz klasami	1289
Kod dekoratorów funkcji	1290
Śledzenie wywołań	1290
Możliwości w zakresie zachowania informacji o stanie	1292
Uwagi na temat klas I — dekorowanie metod klas	1296
Mierzenie czasu wywołania	1301
Dodawanie argumentów dekoratora	1304
Kod dekoratorów klas	1307
Klasy singletona	1307
Śledzenie interfejsów obiektów	1309
Uwagi na temat klas II — zachowanie większej liczby instancji	1312
Dekoratory a funkcje zarządzające	1314
Do czego służą dekoratory (raz jeszcze)	1315
Bezpośrednie zarządzanie funkcjami oraz klasami	1316
Przykład — atrybuty „prywatne” i „publiczne”	1319
Implementacja atrybutów prywatnych	1319
Szczegóły implementacji I	1321
Uogólnienie kodu pod kątem deklaracji atrybutów jako publicznych	1322
Szczegóły implementacji II	1325
Znane problemy	1326
W Pythonie nie chodzi o kontrolę	1333
Przykład — sprawdzanie poprawności argumentów funkcji	1333
Cel	1334
Prosty dekorator sprawdzający przedziały dla argumentów pozycyjnych	1335
Uogólnienie kodu pod kątem słów kluczowych i wartości domyślnych	1337
Szczegóły implementacji	1339
Znane problemy	1342
Argumenty dekoratora a adnotacje funkcji	1344
Inne zastosowania — sprawdzanie typów (skoro nalegamy!)	1346

Podsumowanie rozdziału	1347
Sprawdź swoją wiedzę — quiz	1347
Sprawdź swoją wiedzę — odpowiedzi	1348
40. Metaklasy	1357
Tworzyć metaklasy czy tego nie robić?	1358
Zwiększające się poziomy magii	1359
Język pełen haczyków	1360
Wady funkcji pomocniczych	1361
Metaklasy a dekoratory klas — runda 1.	1363
Model metaklasy	1365
Klasy są instancjami obiektu type	1365
Metaklasy są klasami podrzędnymi klasy type	1368
Protokół instrukcji class	1368
Deklarowanie metaklas	1369
Deklarowanie w wersji 3.x	1370
Deklarowanie w wersji 2.x	1370
Kierowanie metaklas w wersjach 3.x i 2.x	1371
Tworzenie metaklas	1371
Prosta metaklasa	1372
Dostosowywanie tworzenia do własnych potrzeb oraz inicjalizacja	1373
Pozostałe sposoby tworzenia metaklas	1374
Instancje a dziedziczenie	1379
Metaklasa a klasa nadrzędna	1380
Dziedziczenie: pełna historia	1382
Metody metaklas	1388
Metody metaklasy a metody klasy	1388
Przeciążanie operatorów w metodach metaklasy	1389
Przykład — dodawanie metod do klas	1390
Ręczne rozszerzanie	1390
Rozszerzanie oparte na metaklasie	1391
Metaklasy a dekoratory klas — runda 2.	1393
Przykład — zastosowanie dekoratorów do metod	1398
Ręczne śledzenie za pomocą dekoracji	1398
Śledzenie za pomocą metaklas oraz dekoratorów	1399
Zastosowanie dowolnego dekoratora do metod	1400
Metaklasy a dekoratory klas — runda 3. (i ostatnia)	1402
Podsumowanie rozdziału	1404
Sprawdź swoją wiedzę — quiz	1404
Sprawdź swoją wiedzę — odpowiedzi	1405

41. Wszystko, co najlepsze	1407
Paradoks Pythona	1407
„Opcjonalne” cechy języka	1408
Przeciwko niepokojącym usprawnieniom	1408
Złożoność a siła	1409
Prostota a elitarność	1410
Końcowe wnioski	1411
Dokąd dalej?	1411
Na bis: wydrukuj swój certyfikat!	1412
Dodatki	1415
A Instalacja i konfiguracja	1417
B Uruchamianie Pythona 3.x w systemie Windows	1431
C Zmiany w języku Python a niniejsza książka	1445
D Rozwiązania ćwiczeń podsumowujących poszczególne części książki	1457

Pakiety modułów

Dotychczas kiedy importowaliśmy moduły, łądownaliśmy pliki. To typowy model użycia modułów i technika, której w początkach naszej kariery programisty Pythona będziemy używać najczęściej. Importowanie modułów to jednak coś więcej, niż dotychczas sugerowałem.

Poza nazwą modułu w operacji importowania można również wymienić ścieżkę do katalogu. Katalog z kodem Pythona nazywa się *pakiem*, dlatego tego typu operacje importowania znane są jako *importowanie pakietów* (ang. *package import*). W rezultacie importowanie pakietów zamienia katalog z naszego komputera na kolejną przestrzeń nazw Pythona, z atrybutami odpowiadającymi podkatalogom oraz plikom modułów znajdujących się w tym katalogu.

Jest to opcja nieco bardziej zaawansowana, ale udostępniana przez nią hierarchia okazuje się przydatna do organizowania plików w większe systemy i zazwyczaj upraszcza ustawienia ścieżki wyszukiwania modułów. Jak zobaczymy, importowanie pakietów jest czasami wymagane do rozwiązania problemów z operacjami importowania powstających, kiedy na jednym komputerze zainstalowanych jest kilka plików programów o tej samej nazwie.

W niniejszym rozdziale omówimy również wprowadzony w najnowszych wersjach Pythona mechanizm i składnię *importów względnych*, który jest ściśle powiązany z pakietami i modułami. Jak się przekonasz, model ten modyfikuje ścieżki wyszukiwania w wersji 3.x i rozszerza instrukcję `from` w zakresie importowania nazw z pakietów zarówno w wersji 2.x, jak i 3.x. Taki model może sprawić, że importowanie wewnątrz pakietów będzie bardziej wyraźne i zwarte, ale związany jest z pewnymi kompromisami, które mogą wpłynąć na Twoje programy.

Dla użytkowników korzystających z Pythona 3.3 i nowszych wersji omówimy także nowy model *przestrzeni nazw pakietu*, który pozwala pakietom obejmować wiele katalogów i nie wymaga pliku inicjującego. Ten nowy model pakietów jest opcjonalny i może być używany w połączeniu z oryginalnym (lub jak kto woli „zwykłym”) modelem pakietu i rozszerza niektóre podstawowe koncepcje i reguły oryginalnego modelu. Z tego powodu najpierw przeanalizujemy tutaj zwykłe pakiety, a nowy model przestrzeni nazw pakietu przedstawimy jako temat opcjonalny.

Podstawy importowania pakietów

Na poziomie podstawowym importowanie pakietów jest całkiem proste — w miejscu, w którym w instrukcji `import` normalnie wstawiamy nazwę pliku, umieszczamy *ścieżkę* nazw rozdzielonych od siebie kropkami:

```
import dir1.dir2.mod
```

Tak samo wygląda to w przypadku instrukcji `from`.

```
from dir1.dir2.mod import x
```

Ścieżka z kropkami w tych instrukcjach ma odpowiadać ścieżce w systemie plików prowadzącej do pliku `mod.py` (lub podobnego — rozszerzenia mogą być różne). Powyższe instrukcje wskazują zatem, że na naszym komputerze istnieje katalog `dir1`, a w nim podkatalog `dir2` zawierający plik modułu `mod.py` (lub podobny).

Co więcej, takie operacje importowania sugerują, że katalog `dir1` znajduje się w katalogu nadrzędnym `dir0`, dostępnym dla ścieżki wyszukiwania modułów Pythona. Innymi słowy, powyższe instrukcje sugerują, że w systemie plików istnieje struktura przypominająca poniższą (z separatorami w postaci lewych ukośników stosowanych w systemie Windows).

```
dir0\dir1\dir2\mod.py # Lub mod.pyc, mod.so i tak dalej
```

Katalog nadrzędny `dir0` musi być dodany do ścieżki wyszukiwania modułów (o ile nie jest katalogiem głównym dla pliku najwyższego poziomu) — dokładnie tak samo, jakby `dir1` był plikiem modułu.

Mówiąc bardziej formalnie, pierwszy od lewej element ścieżki importu jest nazwą względną w ramach ścieżki wyszukiwania `sys.path`, którą mieliśmy okazję poznać w rozdziale 22. Od tego miejsca do końca ścieżki instrukcje `import` z naszego skryptu w jawny sposób określają ścieżki katalogów prowadzących do modułów.

Pakiety a ustawienia ścieżki wyszukiwania

Jeżeli korzystamy z tej opcji, należy pamiętać, że ścieżki katalogów w instrukcjach `import` mogą być tylko zmiennymi rozdzielonymi kropkami. Nie można w tych instrukcjach użyć żadnej składni ścieżek specyficznej dla określonej platformy, takiej jak `C:\dir1`, `Moje dokumenty`. ↪`dir2` czy `../dir1` — takie ścieżki nie będą działały. Zamiast tego składni specyficznej dla platformy należy użyć w ustawieniach ścieżki wyszukiwania modułów i określić w ten sposób nazwę katalogu nadrzędnego.

W poprzednim przykładzie katalog `dir0` — nazwa katalogu dodawana do ścieżki wyszukiwania modułów — może być dowolnie długą, specyficzną dla platformy ścieżką do katalogu, prowadzącą do katalogu `dir1`. Zamiast używać niepoprawnej instrukcji, takiej jak poniższa:

```
import C:\mycode\dir1\dir2\mod # Błąd — niepoprawna składnia
```

powinieneś dodać katalog `C:\mycode` do zmiennej środowiskowej `PYTHONPATH` lub pliku `.pth`, a następnie wpisać w kodzie programu następujące polecenie:

```
import dir1.dir2.mod
```

W rezultacie wpisy ze ścieżki wyszukiwania modułów będą zawierały *prefiksy* katalogów specyficzne dla platformy i prowadzące do nazw znajdujących się po lewej stronie instrukcji `import` lub `from`. Takie instrukcje importu same z siebie dostarczają pozostałą część ścieżki katalogu w sposób neutralny dla platformy¹.

¹ Składnia z kropkami została wybrana ze względu na swoją neutralność (niezależność od platformy), ale także dlatego, że ścieżki z instrukcji `import` stają się prawdziwymi ścieżkami obiektów zagnieżdżonych. Składnia ta oznacza również, że jeżeli w instrukcjach `import` zapomnisz o pominięciu rozszerzenia `.py`, możesz otrzymać dziwne błędy. Na przykład Python zakłada, że instrukcja `import mod.py` jest operacją importowania ze ścieżką do katalogu, która najpierw załaduje plik `mod.py`, później spróbuje załadować plik `mod\py.py`, a na końcu zwróci

Jeżeli chodzi o proste importowanie plików, nie musisz dodawać katalogu nadrzędnego *dir0* do ścieżki wyszukiwania modułów, jeżeli już tam jest — zgodnie z tym, co pokazywaliśmy w rozdziale 22., będzie to katalog główny pliku najwyższego poziomu, katalog, w którym pracujesz interaktywnie, standardowy katalog biblioteki lub katalog główny instalacji pakietów zewnętrznych. Tak czy inaczej, ścieżka wyszukiwania modułów musi zawierać wszystkie katalogi znajdujące się z lewej strony argumentu instrukcji importowania pakietu kodu.

Pliki pakietów `__init__.py`

Jeżeli zdecydujesz się na importowanie pakietów, musisz pamiętać o jeszcze jednym ograniczeniu, którego należy przestrzegać: przynajmniej do wersji 3.3 Pythona każdy katalog wymieniony w ścieżce instrukcji importowania pakietu musi zawierać plik o nazwie `__init__.py`, w przeciwnym razie operacja importowania zakończy się niepowodzeniem. Oznacza to, że w przykładzie wykorzystanym wyżej oba katalogi, *dir1* oraz *dir2*, muszą zawierać plik o nazwie `__init__.py`. Katalog nadrzędny *dir0* nie musi zawierać tego pliku, ponieważ nie jest on wymieniony w samej instrukcji `import`.

Z formalnego punktu widzenia, jeżeli struktura katalogów wygląda następująco:

```
dir0\dir1\dir2\mod.py
```

a instrukcja `import` ma następującą postać:

```
import dir1.dir2.mod
```

to zastosowanie mają poniższe reguły:

- katalogi *dir1* oraz *dir2* muszą zawierać plik `__init__.py`,
- katalog nadrzędny *dir0* nie musi zawierać pliku `__init__.py`; jeżeli plik ten będzie się w nim znajdował, zostanie zignorowany,
- katalog *dir0*, a nie *dir0\dir1*, musi być umieszczony w ścieżce wyszukiwania modułów `sys.path`.

Aby spełnić dwie pierwsze reguły, deweloperzy pakietów muszą utworzyć odpowiednie pliki, które omówimy już za chwilę. Aby spełnić ostatnią regułę, katalog *dir0* musi być składnikiem automatycznej ścieżki wyszukiwania (czyli musi znajdować się w katalogu domowym użytkownika, katalogu bibliotek lub katalogu *site-packages*) lub musi zostać umieszczony w zmiennej `PYTHONPATH`, pliku *.pth* albo ręcznie dodany do ścieżki `sys.path`.

W efekcie struktura katalogów tego przykładu powinna wyglądać następująco (wcięcia oznaczają zagnieżdżenia katalogów):

```
dir0\                                     # Katalog w ścieżce wyszukiwania modułów
  dir1\
    __init__.py
  dir2\
    __init__.py
    mod.py
```

dość mylący komunikat o błędzie: *No module named py* (brak modułu o nazwie *py*). W wersji 3.3 Pythona ten komunikat o błędzie został poprawiony i obecnie brzmi następująco: *No module named 'mod.py'; mod is not a package* (nie ma modułu o nazwie *mod.py*; *mod* nie jest pakietem).

Pliki `__init__.py` mogą zawierać kod Pythona, podobnie do normalnych plików modułów. Ich nazwy są specjalne, ponieważ zapisany w nich kod jest uruchamiany automatycznie przy pierwszym zaimportowaniu katalogu przez program, a zatem służą przede wszystkim jako punkty zaczepienia do uruchomienia inicjalizacji wymaganych przez pakiet. Pliki te mogą być również zupełnie puste, a czasami pełnią także dodatkowe role, o czym opowiemy w następnym podrozdziale.



Jak zobaczymy pod koniec tego rozdziału, od wersji 3.3 Pythona zniesiono wymóg posiadania przez paczki pliku o nazwie `__init__.py`. W tej wersji i późniejszych katalogi modułów bez takiego pliku można importować jako składające się z jednego katalogu *pakiety przestrzeni nazw*, które działają tak samo, ale nie uruchamiają kodu inicjalizującego. Przed wersją 3.3 i we wszystkich wersjach 2.x pakiety nadal wymagały plików `__init__.py`. Jak pokażemy za chwilę, w wersji 3.3 i późniejszych użycie takich plików wpływa również na zwiększenie wydajności.

Role pliku inicjalizacji pakietu

Pliki `__init__.py` służą jako punkty zaczepienia dla działań odbywających się w czasie inicjalizowania pakietów, deklarują katalogi jako pakiety Pythona, generują przestrzenie nazw dla katalogów i implementują zachowanie instrukcji `from *` (na przykład `from ... import *`), kiedy wykorzystuje się je w połączeniu z importowaniem pakietów.

Inicjalizacja pakietów

Za pierwszym razem, gdy Python importuje coś za pomocą katalogu, automatycznie wykonuje cały kod z pliku `__init__.py` tego katalogu. Z tego powodu pliki te są naturalnym miejscem do wstawienia kodu inicjalizującego stan wymagany przez pakiet, który może na przykład wykorzystać plik inicjalizujący do utworzenia wymaganych plików z danymi czy otwarcia połączenia z bazą danych. Zazwyczaj pliki `__init__.py` nie są przydatne, jeżeli zostaną wykonane bezpośrednio; są one uruchamiane automatycznie przy pierwszym dostępie do pakietu.

Deklaracje użyteczności modułu

Jednym z zadań plików `__init__.py` jest również zadeklarowanie, że dany katalog jest pakietem Pythona. Obecność tych plików zapobiega niezamierzonemu ukrywaniu prawdziwych modułów przez podobne nazwy katalogów, które pojawiają się w ścieżce wyszukiwania modułów. Bez tego zabezpieczenia Python mógłby wybrać katalog, który nie ma nic wspólnego z Twoim kodem, tylko dlatego, że pojawia się wcześniej w ścieżce wyszukiwania. Jak zobaczymy później, pakiety przestrzeni nazw Pythona 3.3 w znacznym stopniu redukują tę rolę, ale uzyskują podobny efekt w sposób algorytmiczny, skanując ścieżkę w poszukiwaniu kolejnych plików.

Inicjalizacja przestrzeni nazw modułu

W modelu importowania pakietów ścieżki katalogów ze skryptu stają się po zaimportowaniu prawdziwymi ścieżkami zagnieżdżonych obiektów. Jak widać w poprzednim przykładzie, po zaimportowaniu wyrażenie `dir1.dir2.mod` działa i zwraca obiekt modułu, którego przestrzeń nazw zawiera wszystkie zmienne przypisane przez plik `__init__.py` katalogu `dir2`. Takie pliki udostępniają przestrzeń nazw dla obiektów modułów tworzonych dla katalogów, które w przeciwnym razie nie miałyby żadnego skojarzonego pliku modułu.

Zachowanie instrukcji `from *`

Jako zaawansowaną opcję możemy wykorzystać listy `__all__` z plików `__init__.py` do zdefiniowania, co jest eksportowane, kiedy katalog importowany jest za pomocą instrukcji `from *`. W pliku `__init__.py` lista `__all__` ma być listą nazw podmodułów, które powinny zostać zaimportowane, kiedy instrukcji `from *` użyjemy na nazwie pakietu (katalogu). Jeżeli lista `__all__` nie zostanie zdefiniowana, instrukcja `from *` nie załaduje automatycznie podmodułów zagnieżdżonych w katalogu. Zamiast tego załaduje tylko zmienne zdefiniowane przez przypisania w pliku `__init__.py` katalogu, w tym wszystkie podmoduły w jawny sposób zaimportowane przez kod tego pliku. Na przykład użycie instrukcji `from submodule import X` w pliku `__init__.py` katalogu sprawia, że zmienna `X` z podmodułu `submodule` będzie dostępna w przestrzeni nazw tego katalogu (dodatkowe zastosowania listy `__all__` omówimy w rozdziale 25.; służy ona również do deklarowania eksportów `from *` z prostych plików).

Pliki `__init__.py` możemy również po prostu pozostawić puste, jeżeli ich rola wykracza poza nasze potrzeby (i szczerze mówiąc, w praktyce te pliki są najczęściej puste). Muszą jednak istnieć, aby importowanie katalogów w ogóle działało.



Nie powinieneś mylić plików `__init__.py` w pakietach z metodami konstruktora klasy `__init__`, które poznamy w następnej części książki. Pierwsze z nich są modułami ładowanymi przez interpreter w ramach importu pakietu, drugie są metodami wywołanymi w celu utworzenia instancji klasy. Oba komponenty mają role inicjujące, ale znacznie różnią się od siebie.

Przykład importowania pakietu

Utwórzmy teraz prawdziwy kod przykładu, o jakim mówiliśmy, w celu pokazania, w jaki sposób działa inicjalizacja plików oraz ścieżek. Poniższe trzy pliki zapisane są w katalogu `dir1` oraz jego podkatalogu `dir2` — w komentarzach zamieszczamy pełne ścieżki do tych plików:

```
# Plik dir1\__init__.py
print('dir1 init')
x = 1

# Plik dir1\dir2\__init__.py
print('dir2 init')
y = 2

# Plik dir1\dir2\mod.py
print('w pliku mod.py')
z = 3
```

Katalog `dir1` będzie tutaj albo bezpośrednim podkatalogiem naszego katalogu roboczego (np. katalogu domowego), albo bezpośrednim podkatalogiem katalogu wymienionego w ścieżce wyszukiwania modułów (czyli z technicznego punktu widzenia w `sys.path`). W obu sytuacjach katalog nadrzędny `dir1` nie musi zawierać pliku `__init__.py`.

Instrukcje `import` wykonują pliki inicjalizacyjne każdego katalogu przy pierwszym przejściu tego katalogu, w miarę jak Python schodzi w dół ścieżki; instrukcje `print` zostały dodane w celu ułatwienia śledzenia sposobu działania plików:

```
C:\code> python
>>> import dir1.dir2.mod
dir1 init
dir2 init
w mod.py
>>>
>>> import dir1.dir2.mod
```

#Uruchamiamy w katalogu nadrzędnym dir1
Pierwszy import wykonuje pliki inicjalizacyjne

Kolejne importy tego nie robią

Tak jak w przypadku plików modułów, zaimportowany już katalog może zostać przekazany do funkcji `reload` w celu wymuszenia ponownego wykonania tego elementu. Jak widać poniżej, funkcja `reload` akceptuje ścieżki z kropkami, by móc przeładować zagnieżdżone katalogi oraz pliki.

```
>>> from imp import reload
>>> reload(dir1)
dir1 init
<module 'dir1' from '.\dir1\__init__.py'>
>>>
>>> reload(dir1.dir2)
dir2 init
<module 'dir1.dir2' from '.\dir1\dir2\__init__.py'>
```

instrukcja from jest niezbędna tylko w wersji 3.x

Po zaimportowaniu ścieżka z instrukcji `import` staje się *ścieżką zagnieżdżonego obiektu* w skrypcie. W kodzie przedstawionym poniżej, `mod` jest obiektem zagnieżdżonym w obiekcie `dir2`, który jest z kolei zagnieżdżony w obiekcie `dir1`.

```
>>> dir1
<module 'dir1' from '.\dir1\__init__.py'>
>>> dir1.dir2
<module 'dir1.dir2' from '.\dir1\dir2\__init__.py'>
>>> dir1.dir2.mod
<module 'dir1.dir2.mod' from '.\dir1\dir2\mod.py'>
```

Tak naprawdę każda nazwa katalogu ze ścieżki staje się zmienną przypisaną do obiektu modułu, którego przestrzeń nazw inicjalizowana jest przez wszystkie przypisania z pliku `__init__.py` tego katalogu. Zmienna `dir1.x` odnosi się do zmiennej `x` przypisanej w pliku `dir1__init__.py`, tak samo jak zmienna `mod.z` odnosi się do zmiennej `z` przypisanej w pliku `mod.py`.

```
>>> dir1.x
1
>>> dir1.dir2.y
2
>>> dir1.dir2.mod.z
3
```

Instrukcja `from` a instrukcja `import` w importowaniu pakietów

Instrukcje `import` mogą być nieco niewygodne w połączeniu z pakietami, ponieważ często musimy w programie wpisywać te same ścieżki. W przykładzie wyżej za każdym razem, gdy chcemy dotrzeć do zmiennej `z`, musimy ponownie wpisać i wykonać pełną ścieżkę od katalogu `dir1`. Jeżeli próbujemy uzyskać bezpośredni dostęp do katalogu `dir2`, otrzymamy błąd.

```
>>> dir2.mod
NameError: name 'dir2' is not defined
>>> mod.z
NameError: name 'mod' is not defined
```

Często w przypadku pakietów wygodniejsze jest zatem skorzystanie z instrukcji `from`, co pozwala uniknąć ponownego wpisywania całych ścieżek przy każdym dostępie do obiektów. Co jednak ważniejsze, jeżeli kiedykolwiek zmienimy strukturę drzewa katalogów, instrukcja

from wymaga uaktualnienia tylko jednej ścieżki w kodzie, podczas gdy import może wiązać się w większą liczbą zmian. Omówione w kolejnym rozdziale rozszerzenie import as może również być pomocne, gdyż podaje krótszy synonim pełnej ścieżki i zapewnia wygodny sposób zmiany nazwy, kiedy taka sama nazwa pojawia się w wielu modułach.

```
C:\code> python
>>> from dir1.dir2 import mod          # Ścieżka podana jedynie tutaj
dir1 init
dir2 init
w pliku mod.py
>>> mod.z                              # Nie powtarzamy ścieżki
3
>>> from dir1.dir2.mod import z
>>> z
3
>>> import dir1.dir2.mod as mod        # Użycie krótszej nazwy (zobacz rozdział 25.)
>>> mod.z
3
>>> from dir1.dir2.mod import z as modz  #To samo, gdy nazwy ze sobą kolidują (zobacz rozdział
25.)
>>> modz
3
```

Do czego służy importowanie pakietów

Osoby zaczynające swoją przygodę z Pythonem powinny przed przejściem do pakietów opanować proste moduły, ponieważ pakiety są już mechanizmem nieco bardziej zaawansowanym. Pełnią jednak użyteczne role, w szczególności w większych programach — sprawiają, że operacje importowania są bardziej informatywne, służą jako narzędzia organizacyjne, upraszczają ścieżkę wyszukiwania modułów i mogą rozwiązać różne niejasności.

Przede wszystkim jednak, ponieważ importowanie pakietów udostępnia pewne informacje o katalogach w plikach programów, ułatwia lokalizację plików i służy także jako narzędzie organizacyjne. Bez ścieżek pakietów często musielibyśmy się odwoływać do ścieżki wyszukiwania pakietów, aby odnaleźć określone pliki. Co więcej, jeżeli zorganizujemy swoje pliki w podkatalogi zgodne z pewnymi obszarami funkcjonalnymi, importowanie pakietów sprawia, że bardziej oczywiste staje się, jaką rolę pełni moduł, dzięki czemu kod jest bardziej czytelny. Na przykład normalne zaimportowanie pliku z katalogu znajdującego się w ścieżce wyszukiwania modułów, takie jak poniższe:

```
import utilities
```

oferuje nam o wiele mniej informacji niż operacja importowania uwzględniająca ścieżkę:

```
import database.client.utilities
```

Importowanie pakietów może również znacznie uprościć nasze ustawienia zmiennej PYTHONPATH oraz plików *.pth*. W rzeczywistości, jeżeli korzystasz z jawnego importowania pakietów i dokonujesz importu pakietów względem wspólnego katalogu głównego, w którym przechowywany jest cały kod Twojego programu, tak naprawdę potrzebujesz tylko jednego wpisu na ścieżce wyszukiwania: wspólnego katalogu głównego. Wreszcie importowanie pakietów służy rozwiązywaniu niejednoznaczności importów poprzez wyraźne określenie, które pliki chcesz zaimportować, i rozwiązuje konflikty, gdy ta sama nazwa modułu pojawia się w więcej niż jednym miejscu. W kolejnym podrozdziale zajmiemy się bardziej szczegółowo tą właśnie rolą.

Historia trzech systemów

Jedyna sytuacja, w której importowanie pakietów jest *wymagane* do rozwiązania niejasności, pojawia się, kiedy na jednym komputerze zainstalowana jest większa liczba programów z plikami noszącymi te same nazwy. Jest to problem instalacyjny, jednak w praktyce może stać się dotkliwy — szczególnie biorąc pod uwagę tendencję programistów do używania prostych i podobnych nazw plików modułów. Aby go zilustrować, zajmiemy się pewnym hipotetycznym scenariuszem wydarzeń.

Załóżmy, że programista tworzy w Pythonie program zawierający plik o nazwie *utilities.py* (ze wspólnym kodem narzędziowym), a także plik najwyższego poziomu *main.py* wykorzystywany przez użytkowników do uruchomienia programu. W całym programie pliki wykorzystują instrukcję `import utilities` do załadowania wspólnego kodu narzędzi. Kiedy program jest dostarczany klientom, stanowi jedno archiwum *.tar* czy *.zip* zawierające wszystkie pliki programu, a po instalacji rozpakowuje wszystkie pliki do jednego katalogu na komputerze docelowym o nazwie *system1*.

```
system1\  
  utilities.py          # Wspólne funkcje i klasy narzędzi  
  main.py              # Uruchamia program  
  other.py             # Importuje utilities w celu załadowania narzędzi
```

Załóżmy teraz, że drugi programista tworzy inny program z plikami o nazwach *utilities.py* oraz *main.py* i ponownie wykorzystuje instrukcję `import utilities` w całym programie w celu załadowania pliku ze wspólnym kodem. Kiedy drugi system zostanie pobrany i zainstalowany na tym samym komputerze co pierwszy, jego pliki zostaną rozpakowane do nowego katalogu o nazwie *system2* na komputerze klienta, tak by nie nadpisały plików o tych samych nazwach z pierwszego systemu.

```
system2\  
  utilities.py          # Wspólne narzędzia  
  main.py              # Uruchamia program  
  other.py             # Importuje narzędzia
```

Jak na razie nie ma żadnych problemów — oba systemy mogą współistnieć i mogą być wykonywane na tym samym komputerze. Tak naprawdę nie musimy nawet konfigurować ścieżki wyszukiwania modułów, by skorzystać z obu programów — ponieważ Python zawsze najpierw przeszukuje katalog główny (czyli katalog zawierający plik najwyższego poziomu), operacje importowania w plikach obu systemów automatycznie zobaczą wszystkie pliki w katalogu danego systemu. Jeżeli na przykład klikniemy plik *system1/main.py*, wszystkie operacje importowania najpierw będą przeszukiwały katalog *system1*. W podobny sposób po uruchomieniu pliku *system2/main.py* jako pierwszy przeszukany zostanie katalog *system2*. Należy pamiętać, że ustawienia ścieżki wyszukiwania modułów są potrzebne tylko wtedy, gdy importujemy pliki pomiędzy katalogami.

Załóżmy jednak, że po zainstalowaniu tych dwóch programów na komputerze decydujemy się użyć jakiejś części kodu z każdego z plików *utilities.py* we własnym programie. W końcu jest to wspólny kod narzędzi, a kod napisany w Pythonie z natury służy do ponownego użycia. W takim przypadku możesz użyć poniższych instrukcji w kodzie pliku utworzonego w trzecim katalogu, tak by załadować jeden z dwóch plików z narzędziami.

```
import utilities  
utilities.func('mielonka')
```

I teraz zaczynamy dostrzegać problem. Aby taki kod działał, musimy ustawić ścieżkę wyszukiwania modułów w taki sposób, by obejmowała ona katalogi zawierające pliki *utilities.py*. Który katalog należy jednak umieścić jako pierwszy — *system1* czy *system2*?

Problemem jest *liniowa* natura ścieżki wyszukiwania, która zawsze jest przeglądana od lewej do prawej strony, więc bez względu na to, jak długo byśmy się nad tym zastanawiali, zawsze otrzymamy plik *utilities.py* z katalogu wymienionego jako pierwszy (bardziej na lewo) w ścieżce wyszukiwania. W takiej postaci nigdy nie będziemy w stanie zaimportować tego pliku z innego katalogu.

Możemy spróbować zmodyfikować `sys.path` w skrypcie przed każdą operacją importowania, ale to dodatkowa operacja, na dodatek bardzo podatna na błędy, a zmiana ustawień zmiennej `PYTHONPATH` przed każdym uruchomieniem programu w Pythonie jest zbyt nużąca i nie pozwala na używanie obu wersji w jednym pliku. Domyślne rozwiązanie sprawia, że jesteśmy w kropce.

Problem ten może rozwiązać właśnie importowanie pakietów. Zamiast instalować programy w niezależnych katalogach wymienionych na ścieżce wyszukiwania modułów osobno, możesz spakować je i zainstalować *w podkatalogach* we wspólnym katalogu głównym. Możemy na przykład zorganizować cały kod tego przykładu w postaci hierarchii katalogów, wyglądającej następująco:

```
root\  
  system1\  
    __init__.py  
    utilities.py  
    main.py  
    other.py  
  system2\  
    __init__.py  
    utilities.py  
    main.py  
    other.py  
  system3\  
    __init__.py      # Tutaj lub w dowolnym innym miejscu  
    myfile.py        # Plik __init__.py jest tutaj potrzebny, gdy pakiet jest importowany w innym miejscu  
                    # Tutaj nasz nowy kod
```

W takiej sytuacji do ścieżki wyszukiwania dodajemy tylko wspólny katalog główny. Jeżeli wszystkie operacje importowania w naszym kodzie są wykonywane względem wspólnego katalogu głównego, możemy zaimportować plik narzędzi z *dowolnego* programu za pomocą importowania pakietu — nazwa katalogu zawierającego plik sprawia, że ścieżka (i tym samym referencja do modułu) staje się unikalna. Tak naprawdę możemy nawet zaimportować *oba* pliki narzędzi w tym samym module, dopóki wykorzystujemy instrukcję `import` i powtarzamy pełną ścieżkę za każdym razem, gdy odwołujemy się do modułów narzędzi.

```
import system1.utilities  
import system2.utilities  
system1.utilities.function('mielonka')  
system2.utilities.function('jajka')
```

Nazwa modułu zawierającego plik sprawia, że referencja do modułu staje się unikalna.

Warto zauważyć, że w przypadku importowania pakietów musimy użyć instrukcji `import` zamiast `from` tylko wtedy, gdy musimy uzyskać dostęp *do tego samego* atrybutu z dwóch lub większej liczby ścieżek. Gdyby nazwa wywoływanej funkcji była w każdej ścieżce inna, można by było użyć instrukcji `from`, co pozwalałoby uniknąć powtarzania pełnej ścieżki za każdym wywołaniem którejś z funkcji, tak jak opisano to wcześniej; do utworzenia unikalnych synonimów nazw możemy również użyć rozszerzenia `as`.

Zwróć również uwagę, że w pokazanej wcześniej hierarchii zainstalowanych plików pliki `__init__.py` zostały dodane do katalogów `system1` oraz `system2`, tak by importowanie pakietów działało, jednak nie znalazły się w *katalogu głównym*. Obecność tych plików jest wymagana jedynie w katalogach wymienionych w instrukcjach `import`. Jak pamiętamy, pliki `__init__.py` są wykonywane automatycznie przez Pythona przy pierwszym importowaniu pakietów.

Z technicznego punktu widzenia w tym przypadku podkatalog `system3` nie musi się znajdować w *katalogu głównym* — dotyczy to jedynie pakietów kodu, z których będziemy importować. Ponieważ jednak nigdy nie wiemy, kiedy nasze własne moduły będą mogły się przydać innym programom, możemy równie dobrze od razu umieścić je we wspólnym katalogu głównym w celu uniknięcia problemów z konfliktami między takimi samymi nazwami w przyszłości.

Wreszcie warto zauważyć, że instrukcje importowania z obu oryginalnych systemów działają bez zmian. Ponieważ najpierw przeszukiwane są ich *katalogi domowe*, dodanie wspólnego katalogu do ścieżki wyszukiwania nie ma znaczenia dla kodu z katalogów `system1` oraz `system2`. Nadal można w nich zastosować polecenie `import utilities` i oczekiwać, że odnajdą w ten sposób własne pliki — choć nie można tego zrobić, jeżeli zostaną użyte jako pakiety w wersji 3.x, co wyjaśnimy w kolejnej sekcji. Co więcej, jeżeli będziemy konsekwentnie rozpakowywać wszystkie programy napisane w Pythonie w jednym katalogu głównym, tak jak zaprezentowano to powyżej, konfiguracja ścieżki staje się banalnie prosta — wystarczy do niej tylko raz dodać wspólny katalog główny.

Warto pamiętać: pakiety modułów

Ponieważ pakiety są standardową częścią Pythona, powszechnie stosowaną praktyką jest to, że większe rozszerzenia firm trzecich są dostarczane jako zestawy katalogów z pakietami, a nie płaskie listy modułów. Na przykład pakiet rozszerzeń `win32all` dla Pythona był jednym z pierwszych, które były udostępniane w postaci pakietów. Wiele jego modułów narzędziowych znajduje się w pakietach importowanych ze ścieżkami. Na przykład, aby załadować narzędzia COM po stronie klienta, powinieneś użyć następującej instrukcji:

```
from win32com.client import constants, Dispatch
```

Przedstawiony wiersz kodu pobiera nazwy z modułu `client` pakietu `win32com` — z podkatalogu instalacyjnego.

Import pakietów jest również wszechobecny w kodzie uruchamianym w implementacji Jython, opartej na języku Java, ponieważ biblioteki Java są również zorganizowane w hierarchie. W najnowszych wydaniach Pythona narzędzia poczty elektronicznej oraz XML są podobnie zorganizowane w podkatalogach pakietów w standardowej bibliotece, a w Pythonie 3.x jeszcze więcej modułów zostało powiązanych w pakiety, w tym narzędzia takie jak graficzny interfejs użytkownika `tkinter`, narzędzia sieciowe HTTP i inne. Przykładowe polecenia pokazane poniżej importują kilka różnych narzędzi bibliotecznych (dla wersji 3.x; sposób użycia w wersji 2.x może się różnić):

```
from email.message import Message
from tkinter.filedialog import askopenfilename
from http.server import CGIHTTPRequestHandler
```

Niezależnie od tego, czy samodzielnie będziesz tworzył katalogi pakietów, czy nie, to prawdopodobnie wcześniej czy później i tak będziesz z nich importował potrzebne Ci narzędzia.

Względne importowanie pakietów

Dotychczas podczas omawiania zagadnień związanych z importowaniem pakietów koncentrowaliśmy się głównie na importowaniu plików *spoza* pakietu. Wewnątrz pakietu importowanie plików tego samego pakietu może korzystać z tej samej składni z pełnymi ścieżkami, co importowanie plików *spoza* pakietu — i jak zobaczymy, czasami nawet powinno. Pliki pakietów mogą jednak również korzystać ze specjalnych, uproszczonych reguł importowania *wewnątrz pakietu*, gdzie zamiast określania pełnej ścieżki do modułu w pakiecie, można zastosować ścieżkę *względną* wewnątrz pakietu.

Sposób działania tego mechanizmu jest zależny od wersji: Python 2.x podczas importowania domyślnie przeszukuje katalogi pakietów, podczas gdy wersja 3.x do importowania z katalogu pakietu wymaga jawnego podania względnej ścieżki pliku w pakiecie. Ta zmiana w wersji 3.x może poprawić czytelność kodu, czyniąc importowanie plików z tego samego pakietu bardziej oczywistym, ale jest jednocześnie niezgodna z wersją 2.x i może spowodować niepoprawne działanie niektórych programów.

Jeżeli rozpoczynasz swoją przygodę z Pythonem od wersji 3.x, powinieneś skupić się na nowej składni i modelu importowania. Jeżeli jednak używałeś już wcześniej starszych wersji Pythona, z pewnością będziesz zainteresowany tym, co zmieniło się w modelu importowania zaimplementowanym w wersji 3.x. Zacznijmy zatem nasze rozważania od tego drugiego tematu.



Jak się przekonasz w tej sekcji, użycie importowania względnego w pakietach może faktycznie *ograniczyć rolę plików*. Krótko mówiąc, nie będzie można ich już używać jako plików programów wykonywalnych w wersjach 2.x i 3.x. Z tego powodu w wielu przypadkach lepszym rozwiązaniem mogą być normalne, pełne ścieżki importowania pakietów. Niemniej ten nowy mechanizm znalazł zastosowanie w wielu programach Pythona i z pewnością zasługuje na to, aby programiści Pythona się z nim zapoznali i lepiej zrozumieli zarówno jego zalety, jak i wady.

Zmiany w Pythonie 3.0

Sposób działania mechanizmu importowania wewnątrz pakietów uległ drobnym zmianom w Pythonie 3.x. Ta zmiana dotyczy wyłącznie importowania plików, które są częścią tego samego pakietu i znajdują się w jego katalogu. Importowanie z innych plików działa tak samo jak dotychczas. W przypadku importowania wewnątrz pakietów w Pythonie 3.x wprowadzono następujące zmiany:

- Zmodyfikowano mechanizm importowania w taki sposób, aby własny katalog pakietu był pomijany. Importowanie sprawdza jedynie ścieżki znajdujące się w ścieżce wyszukiwania `sys.path`. Tego typu import nazywamy *bezwzględnym* (ang. *absolute*).
- Rozszerzono składnię instrukcji `from`, pozwalając na jawne żądanie przeszukiwania ścieżki tylko wewnątrz bieżącego pakietu; możemy to zrobić za pomocą wiodącej kropki w ścieżce pakietu. Tego typu import nazywamy *względnym* (ang. *relative*).

Opisane dwie zmiany obowiązują od wersji 3.x Pythona. Nowa składnia importowania względnego za pomocą `from` jest również dostępna w Pythonie 2.x, ale zmiana mechanizmu domyślnej ścieżki wyszukiwania musi być włączona jako opcja. Włączenie tej opcji może jednak spowodować niepoprawne działanie programów dla wersji 2.x, ale jest dostępne w celu zapewnienia kompatybilności w przód z wersją 3.x.

Skutkiem tej zmiany jest konieczność użycia w Pythonie 3.x (i opcjonalnie w 2.x) specjalnej składni polecenia `from` z kropką wiodącą w ścieżce, pozwalającej na zaimportowanie modułów znajdujących się *w tym samym* pakiecie, chyba że polecenie importu zawiera pełną ścieżkę względem katalogu głównego pakietów w `sys.path` lub polecenia importu są podawane względem zawsze przeszukiwanego katalogu domowego programu (który jest zwykle również bieżącym katalogiem roboczym).

Domyślnie jednak katalog pakietu nie jest automatycznie przeszukiwany, a importowanie wewnątrz pakietu plików znajdujących się bezpośrednio w katalogu pakietu nie powiedzie się bez użycia specjalnej składni polecenia `from`. Jak się niebawem przekonasz, w wersji 3.x może to wpłynąć na sposób strukturyzacji importów lub katalogów modułów przeznaczonych do użycia zarówno w programach najwyższego poziomu, jak i pakietach do importowania. Najpierw jednak przyjrzyjmy się dokładnie, jak to wszystko działa.

Podstawy importowania względnego

Zarówno w Pythonie 3.x, jak i 2.x instrukcje `from` mogą wykorzystywać wiodące kropki (`.`), które sygnalizują, że wymagane moduły są zlokalizowane w tym samym pakiecie (nazywamy to *importem względnym*), a nie w dowolnym miejscu ścieżki wyszukiwania (co nazywamy *importem bezwzględnym*). A dokładniej:

- *Importowanie z użyciem kropek wiodących*: zarówno w Pythonie 3.x, jak i 2.x można użyć wiodącej kropki w instrukcjach `from`, aby wymusić import *względny* w ramach pakietu: tego typu importy wyszukują moduły tylko w katalogu bieżącego pakietu i nie znajdują modułów o tych samych nazwach zapisanych w innych miejscach ścieżki wyszukiwania (`sys.path`). W efekcie moduły danego pakietu mogą przeciążać moduły zewnętrzne.
- *Importowanie bez użycia kropek wiodących*: w Pythonie 2.x importowanie bez wiodącej kropki powoduje wyszukiwanie *w trybie względnym, a następnie bezwzględnym*, z tym, że wyszukiwanie odbywa się w pierwszej kolejności w bieżącym pakiecie. W Pythonie 3.x natomiast importy w ramach pakietu są domyślnie *bezwzględne* — jeżeli nie zastosowano wiodącej kropki, import pomija moduły pakietu i wyszukuje wyłącznie w ścieżce `sys.path`.

Na przykład zarówno w Pythonie 3.x, jak i 2.x można zastosować następującą instrukcję:

```
from . import spam # Import względny w stosunku do pakietu
```

Jej wykonanie spowoduje zaimportowanie modułu o nazwie `spam`, położonego w tym samym katalogu pakietu, co moduł, w którym występuje ta instrukcja. Podobnie instrukcja:

```
from spam import name
```

oznacza „z modułu `spam` znajdującego się w tym samym katalogu zaimportuj zmienną o nazwie `name`”.

Zachowanie instrukcji importu *bez wiodących kropek* jest różne w różnych wersjach Pythona. W 2.x import tego typu domyślnie powoduje wyszukiwanie *względne* (czyli w katalogu bieżącego pakietu), a następnie, jeżeli się ono nie powiedzie, *bezwzględne* w ramach ścieżki wyszukiwania, chyba że na początku pliku importującego jako pierwszą instrukcję wykonywalną umieścimy następujące polecenie:

```
from __future__ import absolute_import # Użyj w wersji 2.x względnego modelu importowania z wersji 3.x
```

Jeżeli takie polecenie występuje, włącza mechanizm wyszukiwania bezwzględnego znanego z wersji 3.x. Po włączeniu tej opcji w wersjach 3.x i 2.x import bez wiodącej kropki w nazwie

modułu zawsze powoduje, że Python pomija względne komponenty ścieżki wyszukiwania importu modułu i zamiast tego szuka wyłącznie w katalogach zawartych w `sys.path`. Na przykład poniższa instrukcja w wersji 3.x spowoduje zaimportowanie modułu `string` ze ścieżki `sys.path`, a nie załaduje modułu `string` zdefiniowanego w bieżącym pakiecie:

```
import string # Pominięcie wersji zdefiniowanej w bieżącym pakiecie
```

Dla kontrastu: bez użycia instrukcji `from __future__` w wersji 2.x Python w pierwszej kolejności podejmie próbę zaimportowania modułu z bieżącego pakietu, jeżeli w pakiecie znajduje się lokalny moduł `string`. Aby uzyskać ten efekt w wersji 3.x oraz w 2.x przy włączonej opcji importu bezwzględnego, należy jawnie zastosować składnię importu względnego:

```
from . import string # Wyszukuje w bieżącym pakiecie
```

Ta instrukcja działa obecnie zarówno w Pythonie 2.x, jak i 3.x. Różnica między wersjami Pythona polega jedynie na tym, że w 3.x zastosowanie tej składni jest *konieczne* do załadowania modułu z bieżącego katalogu, gdy dany plik jest częścią pakietu (chyba że podane zostaną pełne ścieżki pakietów).

Zwróć uwagę, że wiodące kropki mogą być użyte do wymuszenia względnego importowania jedynie w instrukcji `from`, nie w instrukcji `import`. W Pythonie 3.x instrukcja `import nazwa_modułu` zawsze działa w trybie bezwzględnym, czyli pomija bieżący katalog pakietu. W wersji 2.x taka instrukcja nadal wykonuje import względny, najpierw przeszukując katalog pakietu. Instrukcje `from` bez kropek wiodących zachowują się tak samo jak instrukcje `import` — bezwzględnie tylko w 3.x (pomijając katalog pakietu) i względnie, a następnie bezwzględnie w wersji 2.x (najpierw przeszukując katalog pakietu).

Możliwe są również inne względne wzorce odwołań oparte na kropkach. W pliku modułu znajdującym się w katalogu pakietu o nazwie `mypkg` następujące alternatywne formy importu działają tak, jak to opisano w komentarzach:

```
from .string import name1, name2 # Importuje nazwy z mypkg.string
from . import string             # Importuje mypkg.string
from .. import string           # Importuje string z tego samego poziomu, na którym znajduje się mypkg
```

Aby lepiej wyjaśnić działanie tych przykładów i ułatwić zrozumienie przyczyny wdrażania tej dodatkowej złożoności w mechanizmie importowania, musimy dokonać krótkiej dygresji.

Do czego służą importy względne

Oprócz tego, że importowanie wewnątrz tego samego pakietu stało się bardziej przejrzyste, mechanizm importowania względnego został zaprojektowany między innymi po to, aby umożliwić skryptom rozwiązywanie dwuznaczności, które mogą powstawać, gdy plik o tej samej nazwie pojawia się w wielu miejscach ścieżki wyszukiwania modułów. Rozważmy następującą strukturę pakietu:

```
mypkg\
  __init__.py
  main.py
  string.py
```

W ten sposób zdefiniowany jest pakiet `mypkg` zawierający moduły `mypkg.main` i `mypkg.string`. Założmy teraz, że moduł główny próbuje zaimportować moduł o nazwie `string`. W wersji 2.x i nowszych Python zacznie wyszukiwanie w katalogu `mypkg`, realizując *import względny*. Znajdzie zapisany w nim plik o nazwie `print.py` i zaimportuje go, przypisując nazwie `string` w przestrzeni nazw `mypkg.main` pakietu `mypkg`.

Może się jednak okazać, że intencją programisty było zaimportowanie modułu `print` standardowej biblioteki Pythona. Niestety, w starszych wersjach Pythona nie ma prostego sposobu na zignorowanie modułu `mypkg.string` i wymuszenie importu z biblioteki standardowej lub innego modułu znajdującego się w ścieżce wyszukiwania. Co więcej, tego problemu nie możemy również rozwiązać, wykorzystując ścieżki importowania w pakiecie, ponieważ nie możemy zakładać, że układ ścieżek biblioteki standardowej będzie taki sam na każdej maszynie.

Innymi słowy, proste importy w pakietach mogą wprowadzać niejednoznaczności i zwiększać podatność na błędy: w ramach pakietu nie wiadomo, czy instrukcja `import spam` odnosi się do modułu w pakiecie, czy poza nim. W efekcie lokalny moduł lub pakiet może przesłonić (celowo lub przypadkowo) inny, znajdujący się w ścieżce wyszukiwania `sys.path`.

W praktyce użytkownicy Pythona unikają stosowania dla własnych modułów nazw zdefiniowanych w standardowej bibliotece (jeżeli potrzebujesz standardowego modułu `string`, nie powinieneś tworzyć własnego modułu o tej samej nazwie!). Jednak to nie wystarczy, jeżeli w pakiecie zostanie przypadkowo przesłonięty moduł standardowej biblioteki; co więcej, w nowej wersji Pythona mogą pojawić się nowe moduły o nazwach użytych przez nas w programie. Kod wykorzystujący importowanie względne jest też trudniejszy do zrozumienia, ponieważ użytkownik analizujący kod programu może nie mieć pewności co do tego, który pakiet miał być zaimportowany. W kodzie zawsze lepiej jest jednoznacznie dać do zrozumienia, jakie były intencje programisty.

Importowanie względne w wersji 3.x

Aby rozwiązać ten dylemat, mechanizm importowania wewnątrz pakietów zmieniono w Pythonie 3.x i ma teraz charakter wyłącznie bezwzględny (można to również włączyć jako opcję w wersji 2.x). W tym modelu instrukcja `import` w naszym przykładowym pliku `mypkg/main.py` zawsze znajdzie moduł `string` poza pakietem poprzez importowanie bezwzględne ze ścieżki wyszukiwania `sys.path`:

```
import string # Importuje moduł string spoza pakietu (import bezwzględny)
```

Importy z użyciem instrukcji `from` ze ścieżkami bez wiodących kropek również traktowane są jako bezwzględne.

```
from string import name # Importuje name z modułu string poza pakietem
```

Jeżeli chcesz zaimportować moduł zdefiniowany w pakiecie bez podawania jego pełnej ścieżki od katalogu głównego, możesz skorzystać z importowania względnego, umieszczając kropkę w instrukcji `from`:

```
from . import string # Importuje mypkg.string (import względny)
```

Taka forma powoduje zaimportowanie modułu `string` zdefiniowanego w bieżącym pakiecie i jest względnym odpowiednikiem bezwzględnej postaci importu z poprzedniego przykładu (oba polecenia ładują moduł jako całość). W przypadku zastosowania tej składni importu wyszukiwanie jest wykonywane wyłącznie w katalogu pakietu.

Składni importu względnego można użyć również do zaimportowania nazw z modułu:

```
from .string import name1, name2 # Importuje nazwy z mypkg.string
```

Ta instrukcja odwołuje się do modułu `string` zdefiniowanego w bieżącym pakiecie. Jeżeli ten kod wystąpi w module `mypkg.main`, nastąpi `import` nazw `name1` i `name2` z `mypkg.string`.

Pojedyncza kropka w imporcie względnym efektywnie oznacza *bieżący* pakiet, czyli katalog, w którym zapisany jest plik, w którym zadeklarowano import. Dodatkowa kropka spowoduje względny import, rozpoczynając od katalogu *nadrzędnego*, na przykład:

```
from .. import spam # Importuje spam sąsiadujący z mypkg
```

Powyższy kod spowoduje zaimportowanie pakietu `spam` zdefiniowanego na tym samym poziomie, co `mypkg`. Mówiąc bardziej ogólnie, kod znajdujący się w module `A.B.C` może używać dowolnej z następujących form:

```
from . import D # Importuje A.B.D (. oznacza A.B)
from .. import E # Importuje A.E (.. oznacza A)
from .D import X # Importuje A.B.D.x (. oznacza A.B)
from ..E import X # Importuje A.E.x (.. oznacza A)
```

Względne importy a bezwzględne ścieżki pakietów

W module można też wskazać pełną ścieżkę do pakietu, wykorzystując składnię importów bezwzględnych. W poniższym przykładzie pakiet `mypkg` zostanie wczytany ze ścieżki wyszukiwania `sys.path`:

```
from mypkg import string # Importuje mypkg.string (import bezwzględny)
```

Mechanizm ten opiera swoje działanie na konfiguracji i zdefiniowanej kolejności ścieżek, natomiast w przypadku importów względnych nie ma tego typu niejednoznaczności. Co więcej, taki import jak przedstawiony w przykładzie wymaga, aby katalog, w którym zdefiniowany jest pakiet `mypkg`, był zadeklarowany w ścieżce wyszukiwania. Prawdopodobnie dzieje się tak, jeżeli `mypkg` jest katalogiem głównym pakietu (w przeciwnym razie pakiet nie mógłby być użyty z zewnątrz!), ale ten katalog może być zagnieżdżony w znacznie większym drzewie pakietów. Jeżeli `mypkg` nie jest katalogiem głównym pakietu, instrukcje bezwzględnego importu muszą zawierać pełną ścieżkę do pakietu od głównego katalogu w ścieżce wyszukiwania `sys.path`.

```
from system.section.mypkg import string # system znajduje się w katalogu zdefiniowanym w sys.path
```

W przypadku złożonych lub głęboko zagnieżdżonych pakietów pełna ścieżka będzie długa, co może wymagać wpisywania znacznie większej ilości kodu niż w przypadku zastosowania importowania względnego z kropką:

```
from . import string # Składnia importów względnych
```

W tym ostatnim przykładzie pakiet bieżący jest przeszukiwany automatycznie, niezależnie od ustawień ścieżki wyszukiwania, kolejności ścieżki wyszukiwania i zagnieżdżania katalogu. Z drugiej strony bezwzględna forma pełnej ścieżki będzie działać bez względu na to, w jaki sposób plik jest używany — jako część programu czy pakietu — o czym opowiemy już w kolejnym podrozdziale.

Zasięg importów względnych

Importy względne mogą przy pierwszym kontakcie wydać się dość skomplikowanym zagadnieniem, ale wszystko znacznie się upraszcza po poznaniu pewnych podstawowych zasad:

- **Importy względne są stosowane wyłącznie wewnątrz pakietów.** Pamiętaj, że zmiana ścieżki wyszukiwania modułu dotyczy tylko instrukcji importowania w plikach modułu używanych jako część pakietu — czyli inaczej mówiąc, dotyczy importowania *wewnątrz pakietu*. Normalne importy w plikach, które nie są częścią pakietu, nadal działają dokładnie

tak, jak opisano wcześniej, automatycznie przeszukując najpierw katalog zawierający skrypt najwyższego poziomu.

- **Względne importy stosuje się wyłącznie w instrukcji `from`.** Pamiętaj też, że nowa składnia importowania dotyczy tylko instrukcji `from` i nie ma zastosowania do instrukcji `import`. Importy względne rozpoznawane są po tym, że po słowie kluczowym `from` następuje jedna lub więcej kropek. Nazwy modułów zawierające kropki, ale bez kropki wiodącej są traktowane jako zwykły import pakietów, a nie import względny.

Innymi słowy: względne importy z pakietów w wersji 3.x to w zasadzie rezygnacja ze stosowanej w Pythonie 2.x reguły wyszukiwania modułów w pakietach oraz dodanie składni wymuszającej wyszukiwanie względne w pakiecie. Jeżeli pisałeś kod dla Pythona tak, aby nie korzystać z wyszukiwania względnego, charakterystycznego dla wersji 2.x (na przykład zawsze podawałeś pełną ścieżkę od katalogu głównego do modułu), to zmiany wprowadzone w Pythonie 3.x raczej nie będą stanowiły zagrożenia z punktu widzenia kompatybilności. Jeżeli tego nie zrobiłeś, musisz zaktualizować pliki pakietu, aby korzystać z nowej składni polecenia `from` dla lokalnych plików pakietów lub używać pełnych ścieżek bezwzględnych.

Podsumowanie reguł wyszukiwania modułów

W przypadku pakietów i importowania względnego reguły wyszukiwania modułów w Pythonie 3.x, które omawialiśmy do tej pory, można streścić w następujący sposób:

- Podstawowe moduły z prostymi nazwami (np. `A`) są wyszukiwane w każdym katalogu z listy `sys.path`, od lewej do prawej. Ta lista jest budowana z domyślnych ustawień systemowych i uzupełniana o ustawienia konfigurowane przez użytkownika.
- Pakiety są po prostu katalogami zawierającymi moduły Pythona oraz specjalny plik `__init__.py`, który umożliwia stosowanie w importach ścieżki typu `A.B.C`. W takim imporcie katalog `A` powinien znajdować się w ścieżce wyszukiwania `sys.path`, `B` to podkatalog katalogu `A`, natomiast `C` jest modułem lub inną nazwą importowaną z `B`.
- W plikach pakietów zwykle instrukcje `import` oraz `from` stosują tę samą regułę z użyciem ścieżki wyszukiwania `sys.path`. Importy w pakietach wykorzystujące instrukcję `from` oraz ścieżkę modułu rozpoczynającą się od *kropki* stosują specjalną regułę importów względnych, to znaczy nazwa jest wyszukiwana wyłącznie w odniesieniu do pakietu, a wyszukiwanie w ścieżce `sys.path` nie jest wykonywane. Na przykład w instrukcji `from . import A` wyszukiwanie modułu `A` będzie realizowane wyłącznie w katalogu zawierającym plik, w którym zdefiniowano tę instrukcję.

Python 2.x działa tak samo, z tym wyjątkiem, że normalne importowanie ze ścieżką bez kropek wiodących również automatycznie najpierw przeszukuje *katalog pakietów*, zanim przejdzie do `sys.path`.

Podsumowując, importowanie w Pythonie wybiera między *względnymi* (w zawierającym katalogu) i *bezwzględnymi* (w katalogu ze ścieżki wyszukiwania `sys.path`) operacjami w następujący sposób:

Importowanie z kropką: `from . import m, from .m import x`
Jest *względne* zarówno w wersji 2.x, jak i 3.x.

Importowanie bez kropki: `import m, from m import x`
Jest *najpierw względne*, a *następnie bezwzględne* w wersji 2.x, a *bezwzględne* tylko w wersji 3.x.

Jak zobaczymy później, Python 3.3 dodaje kolejny element do modułów — *pakiety przestrzeni nazw* (ang. *namespace packages*) — który jest w dużej mierze odmienny od historii importowania względnego, tutaj omawianej. Ten nowszy model również obsługuje importowanie względne wewnątrz pakietów i jest po prostu innym sposobem na zbudowanie pakietu. Pakiety przestrzeni nazw usprawniają procedurę wyszukiwania importu, umożliwiając rozłożenie zawartości pakietu na wiele prostych katalogów, ale później pakiet jako całość zachowuje się tak samo w kwestii względnych reguł importu.

Importy względne w działaniu

Wystarczy tej teorii, pokażemy zatem kilka prostych przykładów, aby zademonstrować w praktyce koncepcję importów względnych.

Importowanie spoza pakietów

Przede wszystkim, jak już wspominaliśmy wcześniej, mechanizm importów względnych nie ingeruje w możliwość importowania spoza pakietów. Dzięki temu poniższy kod importujący moduł `string` standardowej biblioteki Pythona zadziała zgodnie z oczekiwaniami:

```
C:\code> c:\Python33\python
>>> import string
>>> string
<module 'string' from 'C:\Python33\lib\string.py'>
```

Jeżeli jednak do katalogu, w którym pracujemy, dodamy moduł o nazwie `string`, to załadowany zostanie ten moduł, ponieważ na pierwszym miejscu w ścieżce wyszukiwania znajduje się bieżący katalog roboczy (ang. *CWD* — *current working directory*).

```
# code\string.py
print('string' * 8)

C:\code> c:\Python33\python
>>> import string
stringstringstringstringstringstringstringstring
>>> string
<module 'string' from '.\string.py'>
```

Innymi słowy, zwykłe importy nadal są względne w stosunku do katalogu „domowego” (czyli katalogu, w którym zapisany jest skrypt, lub katalogu, z którego został uruchomiony). W rzeczywistości składnia importów względnych nie jest dozwolona w kodzie znajdującym się w pliku niebędącym częścią pakietu.

```
>>> from . import string
SystemError: Parent module '' not loaded, cannot perform relative import
```

W tej sekcji kod wprowadzany w sesji interaktywnej zachowuje się tak samo, jakby był uruchamiany w skrypcie najwyższego poziomu, ponieważ pierwszym wpisem w ścieżce `sys.path` jest albo interaktywny katalog roboczy, albo katalog zawierający plik najwyższego poziomu. Jedyna różnica polega na tym, że pierwszy element ścieżki `sys.path` jest katalogiem bezwzględnym, a nie pustym ciągiem znaków:

```
# code\main.py
import string
print(string)

# Ten sam kod, ale w pliku

C:\code> C:\python33\python main.py
stringstringstringstringstringstringstringstring
<module 'string' from 'C:\code\string.py'>

# Wyniki w wersji 2.x będą takie same
```

Jak widać, próba wykonania polecenia `from . import string` w pliku niebędącym częścią pakietu kończy się niepowodzeniem, tak samo jak w przypadku sesji interaktywnej.

Importy wewnątrz pakietów

Usuń teraz lokalny moduł `string`, który zapisaliśmy w bieżącym katalogu roboczym, i zbuduj tam katalog pakietu zawierający dwa moduły, w tym wymagany, ale pusty plik `code\pkg__init__.py`. Główne katalogi pakietów omawianych w tej sekcji znajdują się w bieżącym katalogu roboczym, dodawany automatycznie do ścieżki `sys.path`, więc nie musimy ustawiać zmiennej `PYTHONPATH`. Ze względu na oszczędność miejsca w dużej mierze pominiemy też puste pliki `__init__.py` i większość komunikatów o błędach (a użytkownicy systemów innych niż Windows będą musieli samodzielnie przetłumaczyć pokazane niżej polecenia powłoki na swoją platformę):

```
C:\code> del string*                # w wersjach 3.2+ usuwamy pliki kodu bajtowego __pycache__\string*
C:\code> mkdir pkg
C:\code> notepad pkg\__init__.py

# code\pkg\spam.py
import eggs                          # <== Działa w wersji 2.x, ale nie w 3.x!
print(eggs.x)

# code\pkg\eggs.py
X = 99999
import string
print(string)
```

W pierwszym module tego pakietu próbujemy zaimportować drugi za pomocą zwykłej instrukcji `import`. W wersji 2.x Python potraktuje takie polecenie jako `import` względny, ale w wersji 3.x jako bezwzględny, stąd w tym drugim przypadku próba wykonania tego polecenia zakończy się niepowodzeniem. Innymi słowy, w wersji 2.x najpierw przeszukiwany jest pakiet modułu, w którym wykonywany jest `import`, ale w wersji 3.x tak się nie dzieje. Jest to jeden ze szczegółów implementacji Pythona 3.x, który *nie jest kompatybilny* z poprzednimi wersjami, i należy mieć ten fakt na uwadze.

```
C:\code> c:\Python27\python
>>> import pkg.spam
<module 'string' from 'c:\Python27\lib\string.pyc'>
99999

C:\code> c:\Python33\python
>>> import pkg.spam
ImportError: No module named 'eggs'
```

Aby nasze moduły działały prawidłowo zarówno w wersji 2.x, jak i 3.x, powinniśmy zmodyfikować instrukcję importu w pierwszym z plików w taki sposób, by wykorzystać składnię importów względnych i wskazać Pythonowi, aby moduł `eggs` szukał również w katalogu pakietu (dla wersji 3.x):

```
# code\pkg\spam.py
from . import eggs                    # <== użycie względnych importów w 2.x i 3.x
print(eggs.x)

# code\pkg\eggs.py
X = 99999
import string
print(string)

C:\code> c:\Python27\python
>>> import pkg.spam
```



```

<module 'string' from 'c:\Python27\lib\string.pyc'>
99999

C:\code> c:\Python33\python
>>> import pkg.spam
<module 'string' from 'c:\Python33\lib\string.py'>
99999

```

Importy są nadal względne w stosunku do bieżącego katalogu roboczego

Zwróć uwagę na to, że moduły w pakietach ciągle mają dostęp do modułów biblioteki standardowej, takich jak `string`, jednak ich normalne importy są nadal względne w stosunku do ścieżki wyszukiwania. W rzeczywistości, jeżeli do katalogu roboczego dodamy moduł `string`, to przy próbie importu zostanie załadowany właśnie on, nie moduł biblioteki standardowej. Choć w wersji 3.x możesz pominąć katalog pakietu za pomocą importu bezwzględnego, nie ma możliwości pominięcia katalogu głównego programu, który importuje taki pakiet:

```

# code\string.py
print('string' * 8)

# code\pkg\spam.py
from . import eggs
print(eggs.x)

# code\pkg\eggs.py
X = 99999
import string # <== Znajduje moduł string w katalogu roboczym, nie w bibliotece standardowej!
print(string)

C:\code> c:\Python33\python # W wersji 2.x wynik będzie taki sam
>>> import pkg.spam
stringstringstringstringstringstringstringstring
<module 'string' from '.\string.py'>
99999

```

Użycie importów względnych i bezwzględnych

Aby pokazać, w jaki sposób zasady wyszukiwania modułów wpływają na użycie modułów biblioteki standardowej, ponownie zresetujemy nasz pakiet. Usuń lokalny moduł `string` i utwórz nowy, ale wewnątrz pakietu.

```

C:\code> del string* # w wersjach 3.2+ usuwamy pliki kodu bajtowego __pycache__\string*

# code\pkg\spam.py
import string # <== Względny w wersji 2.x, bezwzględny w wersji 3.x
print(string)

# code\pkg\string.py
print('Ni' * 8)

```

To, która wersja modułu `string` zostanie zaimportowana, zależy od użytej wersji Pythona. Jak już pokazywaliśmy, wersja 3.x `import` w pierwszym pliku traktuje jako bezwzględny, pomijając moduły pakietu, ale w 2.x tak się nie dzieje — to kolejny przykład braku *kompatybilności wstecznej* wersji 3.x.

```

C:\code> c:\Python33\python
>>> import pkg.spam
<module 'string' from 'C:\Python33\lib\string.py'>

```

```
C:\code> c:\Python27\python
>>> import pkg.spam
NiNiNiNiNiNiNiNiNi
<module 'pkg.string' from 'pkg\string.py'>
```

Użycie składni importów względnych w wersji 3.x wymusza przeszukiwanie pakietu podobnie jak w wersji 2.x, ale dzięki możliwości użycia w wersji 3.x importów bezwzględnych lub względnych mamy pełną kontrolę nad tym, który pakiet zostanie zaimportowany. W rzeczywistości *to właśnie jest przyczyną wprowadzenia takiej zmiany w Pythonie 3.x*.

```
# code\pkg\spam.py
from . import string          # <== Import względny zarówno w wersji 2.x, jak i 3.x
print(string)
```

```
# code\pkg\string.py
print('Ni' * 8)
```

```
C:\code> c:\Python33\python
>>> import pkg.spam
NiNiNiNiNiNiNiNiNi
<module 'pkg.string' from '.\pkg\string.py'>
```

```
C:\code> c:\Python27\python
>>> import pkg.spam
NiNiNiNiNiNiNiNiNi
<module 'pkg.string' from 'pkg\string.py'>
```

Importy względne przeszukują tylko pakiety

Zwróć uwagę na to, że składnia importów względnych jest w rzeczywistości *deklaracją wiązania*, nie jedynie właściwością. Jeżeli usuniemy plik *string.py* i cały powiązany z nim kod bajtowy z naszego przykładu, import względny zadeklarowany w pliku *spam.py* nie uda się ani w wersji 3.x, ani w 2.x — Python nie podejmie próby zaimportowania modułu *string* ze standardowej ścieżki wyszukiwania (czyli z biblioteki standardowej ani z jakiegokolwiek innej zdefiniowanej w ścieżce).

```
# code\pkg\spam.py
from . import string          # <== Nie działa ani w wersji 2.x, ani w 3.x; w pakiecie nie ma pliku
string.py!
```

```
C:\code> del pkg\string*
```

```
C:\code> C:\python33\python
>>> import pkg.spam
ImportError: cannot import name string
```

```
C:\code> C:\python27\python
>>> import pkg.spam
ImportError: cannot import name string
```

Moduły wskazywane przez importy względne muszą istnieć w katalogu pakietu.

Importy są nadal względne w stosunku do katalogu roboczego (cd.)

Importy bezwzględne pozwalają pominąć moduły pakietów, ale nadal są zależne od elementów ścieżki *sys.path*. W ostatnim teście zdefiniujemy dwa własne moduły *string*. Zapiszemy je w ten sposób, aby jeden z modułów był w pakiecie, drugi w bieżącym katalogu roboczym; trzeci moduł znajduje się w bibliotece standardowej Pythona.

```

# code\string.py
print('string' * 8)

# code\pkg\spam.py
from . import string           # <== Względny zarówno w wersji 2.x, jak i 3.x
print(string)

# code\pkg\string.py
print('Ni' * 8)

```

Gdy zaimportujemy moduł `string` z użyciem składni importów względnych, w obu wersjach Pythona otrzymamy moduł zapisany w pakiecie, dokładnie tak, jak się tego spodziewamy:

```

C:\code> c:\Python33\python    # Taki sam rezultat w 2.x
>>> import pkg.spam
NiNiNiNiNiNiNiNiNiNi
<module 'pkg.string' from '.\pkg\string.py'>

```

Gdy użyjemy składni importów bezwzględnych, wynik importu będzie ponownie zależny od wersji Pythona: w wersji 2.x import ten będzie względny, a w wersji 3.x „bezwzględny”, co w tym przypadku oznacza pominięcie wersji zdefiniowanej w pakiecie i użycie pakietu zdefiniowanego w ścieżce roboczej (wersja z biblioteki standardowej *nie będzie użyta*).

```

# code\string.py
print('string' * 8)

# code\pkg\spam.py
import string                 # <== Import względny w 2.x, bezwzględny w 3.x: katalog roboczy!
print(string)

# code\pkg\string.py
print('Ni' * 8)

C:\code> c:\Python33\python
>>> import pkg.spam
stringstringstringstringstringstringstringstringstring
<module 'string' from '.\string.py'>

C:\code> c:\Python27\python
>>> import pkg.spam
NiNiNiNiNiNiNiNiNiNi
<module 'pkg.string' from 'pkg\string.pyc'>

```

Jak widzimy, pakiety mogą deklarować import modułów z lokalnych pakietów, ale importy pozostają względne w stosunku do ścieżki wyszukiwania. W tym przypadku plik zdefiniowany w katalogu roboczym programu przesłania moduł biblioteki standardowej o tej samej nazwie. Zmiany mechanizmu wyszukiwania modułów wprowadzone w wersji 3.x realizują jedynie możliwość wyboru modułu z bieżącego pakietu lub spoza niego (w ramach importów względnych lub bezwzględnych). Jednak mechanizm importu jest zależny od środowiska, w którym jest wykonywany, bezwzględne importy w wersji 3.x nadal nie są zabezpieczeniem przed efektami ubocznymi przesłonięcia nazw modułów przez inne zapisane w ścieżce wyszukiwania.

Warto przeprowadzić kilka eksperymentów z tymi przykładami w celu lepszego zrozumienia zasad importów. W rzeczywistości problemy z importami nie są tak powszechne, jak można by się obawiać, wnioskując z przykładów: importy można ustrukturalizować, ścieżkę wyszukiwania można modyfikować do własnych potrzeb, aby importy działały dokładnie tak, jak tego oczekujemy. Należy jednak mieć na uwadze, że importy w bardziej skomplikowanych konfiguracjach systemowych mogą być zależne od kontekstu, a na protokół importu modułów ma wpływ jakość projektu biblioteki aplikacji.

Pułapki związane z importem względnym w pakietach: zastosowania mieszane

Teraz gdy znasz już sposoby działania importów względnych w pakietach, powinieneś również zapamiętać, że nie zawsze są one najlepszym wyborem. Bezwzględny import pakietów, z pełną ścieżką do katalogu w katalogu `sys.path`, nadal jest najczęściej preferowanym rozwiązaniem, zarówno w przypadku niejawnego importu względnego w Pythonie 2.x, jak i jawnej składni kropkowej importu względnego w Pythonie 2.x i 3.x. Początkowo może się to wydawać mało istotne, ale najprawdopodobniej szybko docenisz ważność tego zagadnienia po rozpoczęciu samodzielnego kodowania własnych pakietów.

Jak widzieliśmy, składnia importów względnych w Pythonie 3.x i domyślna reguła wyszukiwania bezwzględnego wymuszają jawne importowanie wewnątrz pakietów, dzięki czemu kod staje się bardziej czytelny i łatwiejszy do zrozumienia, a co więcej, umożliwia także dokonywanie wyraźnego wyboru w niektórych scenariuszach konfliktu nazw. Są jednak również dwa główne następstwa tego modelu, o których należy pamiętać:

- Zarówno w Pythonie 3.x, jak i 2.x użycie instrukcji importu względnego dla pakietu niejawnie wiąże plik z katalogiem pakietu i rolę, wykluczając tym samym jego użycie w inny sposób.
- W Pythonie 3.x nowa zmiana reguły wyszukiwania względnego oznacza, że plik nie może już służyć jednocześnie jako moduł skryptu i moduł pakietu tak łatwo, jak to jest możliwe w wersji 2.x.

Przyczyny tych ograniczeń są nieco subtelne, ale dość oczywiste, biorąc pod uwagę następujące fakty:

- Ani wersja 3.x, ani 2.x Pythona nie pozwalają na używanie względnej instrukcji `from .`, chyba że plik importujący jest częścią pakietu (inaczej mówiąc, moduł będący częścią pakietu jest importowany z innego miejsca pakietu).
- Podczas importowania Python 3.x nie przeszukuje własnego katalogu modułów pakietu, chyba że użyjemy względnej instrukcji `from .` (lub moduł znajduje się w bieżącym katalogu roboczym albo katalogu domowym głównego skryptu).

Korzystanie z importu względnego uniemożliwia tworzenie pakietów, które służą zarówno jako programy wykonywalne, jak i pakiety do importu z zewnątrz. Co więcej, niektóre pliki również nie mogą już służyć jednocześnie jako moduły skryptów i moduły pakietów. Jeżeli chodzi o instrukcje importu, reguły wyglądają następująco — pierwszy przykład dotyczy trybu *pakiet* w obu Pythonach, a drugi dotyczy trybu *program* tylko w wersji 3.x:

```
from . import mod          # Niedozwolone w trybie bez pakietu zarówno w wersji 2.x, jak i 3.x
import mod                # W trybie pakietu nie przeszukuje własnego katalogu pliku (w wersji 3.x)
```

W efekcie w przypadku plików, które mają być używane w wersji 2.x lub 3.x, może być konieczne wybranie jednego trybu użycia — *pakietu* (z importem względnym) lub *programu* (z prostym importami) i wyodrębnienie prawdziwych plików modułów w osobnym podkatalogu, oddzielnym od plików skryptów najwyższego poziomu.

Alternatywnie możesz spróbować ręcznie wprowadzać zmiany w ścieżce `sys.path` (zadanie niewdzięczne i podatne na błędy) lub zawsze używać pełnych ścieżek pakietów w importach

bezwzględnych zamiast składni względnej lub prostych importów i zakładać, że katalog główny pakietu znajduje się w ścieżce wyszukiwania modułów:

```
from system.section.mypkg import mod # Działa zarówno w trybie program, jak i pakiet
```

Ze wszystkich opisanych schematów ten ostatni — import z pełnymi ścieżkami pakietu — może być najbardziej przenośny i funkcjonalny, ale musimy przejść do bardziej konkretnego kodu, aby przekonać się dlaczego.

Problem

Na przykład w Pythonie 2.x często stosuje się ten sam *pojedynczy katalog* jako program i pakiet, używając normalnego importu bez kropek. W trybie programu importy opierają się na katalogu domowym programu (skrypty), a w trybie pakietu (importy wewnątrz pakietu) najpierw przeszukiwany jest względny katalog pakietu, a dopiero potem ścieżka bezwzględna. Nie działa to jednak w wersji 3.x — w trybie pakietu zwykłe importowanie nie ładuje już modułów znajdujących się w tym samym katalogu, chyba że katalog ten jest głównym katalogiem pakietu lub bieżącym katalogiem roboczym (i dlatego znajduje się na ścieżce `sys.path`).

Oto jak to wygląda w działaniu, z minimalną ilością kodu (dla zwięzłości w tej sekcji ponownie pomijam pliki `__init__.py` wymagane dla pakietów przed wersją 3.3 Pythona, a dla wprowadzenia małego urozmaicenia używam programu uruchamiającego z systemu Windows, opisanego w dodatku B):

```
# code\pkg\main.py
import spam

# code\pkg\spam.py
import eggs # <== Działa, jeżeli jest w katalogu "." =, czyli katalogu domowym
głównego skrypty

# code\pkg\eggs.py
print('Eggs' * 4) # Ale plik nie zostanie załadowany, jeżeli jest używany jako pakiet w 3.x!

c:\code> python pkg\main.py # OK jako program, zarówno w 2.x, jak i w 3.x
EggsEggsEggsEggs
c:\code> python pkg\spam.py
EggsEggsEggsEggs

c:\code> py 2 # OK jako pakiet w 2.x; wyszukiwanie względne, a później bezwzględne
>>> import pkg.spam # 2.x: proste importy przeszukują najpierw katalog pakietów
EggsEggsEggsEggs

C:\code> py 3 # Ale wersja 3.x nie potrafi odnaleźć tutaj pliku; pozostaje tylko import
bezwzględny # 3.x: proste importy przeszukują tylko bieżący katalog roboczy plus
>>> import pkg.spam # 3.x: proste importy przeszukują tylko bieżący katalog roboczy plus
ścieżkę sys.path
ImportError: No module named 'eggs'
```

Następnym krokiem mogłoby być dodanie *importu względnego* do użytku w wersji 3.x, ale tutaj to nie pomoże. W poniższym przykładzie zachowujemy pojedynczy katalog zarówno dla głównego skrypty, jak i modułów pakietu oraz dodajemy wymagane kropki — takie rozwiązanie działa zarówno w wersji 2.x, jak i 3.x, gdy katalog jest importowany jako pakiet, ale kończy się niepowodzeniem, gdy jest używany jako katalog programu (w tym podczas próby bezpośredniego uruchomienia modułu jako skrypty):

```
# code\pkg\main.py
import spam
```

```

# code\pkg\spam.py
from . import eggs # <== Nie jest pakietem, jeżeli znajduje się tutaj skrypt główny

# code\pkg\eggs.py
print('Eggs' * 4)

c:\code> python # OK jako pakiet, ale nie jako program (w wersjach 3.x i 2.x)
>>> import pkg.spam
EggsEggsEggsEggs

c:\code> python pkg\main.py
SystemError: ... cannot perform relative import
c:\code> python pkg\spam.py
SystemError: ... cannot perform relative import

```

Rozwiązanie nr 1: podkatalogi pakietów

W przypadku takiego mieszanego zastosowania jednym z rozwiązań jest izolacja wszystkich plików oprócz głównego skryptu w osobnym *podkatalogu* — w ten sposób importowanie modułów wewnątrz pakietu nadal działa we wszystkich Pythonach, możesz używać katalogu nadrzędnego jako samodzielnego programu, a katalog zagnieżdżony nadal może służyć jako pakiet do użytku dla innych programów:

```

# code\pkg\main.py
import sub.spam # <== Działa, jeżeli przeniesiemy moduły do podkatalogu „poniżej” pliku
głównego

# code\pkg\sub\spam.py
from . import eggs # Importy względne wewnątrz pakietu teraz działają poprawnie (moduły
w podkatalogach)

# code\pkg\sub\eggs.py
print('Eggs' * 4)

c:\code> python pkg\main.py # Z głównego skryptu — ten sam rezultat zarówno w wersji 2.x, jak i 3.x
EggsEggsEggsEggs

c:\code> python # Z innych programów — ten sam rezultat zarówno w wersji 2.x, jak i 3.x
>>> import pkg.sub.spam
EggsEggsEggsEggs

```

Potencjalnym minusem takiego schematu jest to, że nie będzie można uruchamiać modułów pakietów bezpośrednio w celu sprawdzenia ich za pomocą odpowiedniego kodu testującego, choć zamiast tego wszelkie testy można osobno umieścić w katalogu nadrzędnym:

```

c:\code> py 3 pkg\sub\spam.py # Poszczególne moduły nie mogą być uruchamiane indywidualnie, np. do
testów
SystemError: ... cannot perform relative import

```

Rozwiązanie 2: import bezwzględny z użyciem pełnej ścieżki

Alternatywnie użycie *importu* z podaniem pełnej ścieżki również rozwiązałoby nasz problem — taki scenariusz wymaga co prawda, aby katalog nadrzędny dla głównego katalogu pakietu był w ścieżce wyszukiwania, ale w praktyce nie jest to nic nadzwyczajnego dla rzeczywistych pakietów oprogramowania. Większość pakietów Pythona albo wymaga takiego ustawienia, albo zarządza jego automatyczną obsługą za pomocą narzędzi instalacyjnych (takich jak *distutils*, które pozwalają na przechowywanie pakietu w katalogu znajdującym się w domyślnej ścieżce wyszukiwania modułów, takim jak katalog główny pakietów witryny; więcej szczegółowych informacji na ten temat znajdziesz w rozdziale 22.):

```

# code\pkg\main.py
import spam

# code\pkg\spam.py
import pkg.eggs          # <== Pełne ścieżki działają we wszystkich przypadkach; 2.x+3.x

# code\pkg\eggs.py
print('Eggs' * 4)

c:\code> set PYTHONPATH=C:\code
c:\code> python pkg\main.py    # Z głównego skryptu — ten sam rezultat zarówno w wersji 2.x, jak i 3.x
EggsEggsEggsEggs

c:\code> python                # Z innych programów — ten sam rezultat zarówno w wersji 2.x, jak i 3.x
>>> import pkg.spam
EggsEggsEggsEggs

```

W przeciwieństwie do rozwiązania z podkatalogiem importy bezwzględne pełnej ścieżki, takie jak te pokazane powyżej, pozwalają również na samodzielne uruchamianie modułów w celu przetestowania:

```

c:\code> python pkg\spam.py    # Poszczególne moduły mogą być uruchamiane indywidualnie w 2.x i 3.x
EggsEggsEggsEggs

```

Przykład: aplikacja z kodem autotestu modułu (wprowadzenie)

Na koniec przedstawimy jeszcze jeden przykład typowego problemu i jego pełną ścieżkę rozwiązania. Wykorzystuje ona powszechną technikę, którą omówimy w następnym rozdziale, ale sama koncepcja jest wystarczająco prosta, aby pokazać ją już tutaj (choć możesz równie dobrze wrócić do tego przykładu nieco później).

Rozważ następujące dwa moduły w katalogu pakietu, z których drugi zawiera kod *autotestu*. Upraszczając, wartością atrybutu `__name__` tego modułu jest ciąg `__main__`, gdy jest uruchamiany jako skrypt najwyższego poziomu, ale nie podczas importowania, co pozwala na używanie tego modułu zarówno jako modułu, jak i skryptu:

```

# code\dualpkg\m1.py
def somefunc():
    print('m1.somefunc')

# code\dualpkg\m2.py
...tutaj import m1...          # Zamień ten wiersz na odpowiednie polecenie importu

def somefunc():
    m1.somefunc()
    print('m2.somefunc')

if __name__ == '__main__':
    somefunc()                 # Autotest lub kod skryptu najwyższego poziomu

```

Drugi moduł musi zaimportować ten pierwszy, w którym pojawia się wiersz zastępczy `... tutaj import m1...`. Zastąpienie tego wiersza względną instrukcją importu działa, gdy plik jest używany jako pakiet, ale nie jest dozwolone w trybie skryptu ani w wersji 2.x, ani w 3.x (wyniki i komunikaty o błędach zostały tutaj pominięte ze względu na oszczędność miejsca; zobacz plik `dualpkg\results.txt` w przykładach książki dla pełnego zestawu komunikatów):

```

# code\dualpkg\m2.py
from . import m1

```

```

c:\code> py 3
>>> import dualpkg.m2 # OK
C:\code> py 2
>>> import dualpkg.m2 # OK

c:\code> py 3 dualpkg\m2.py # Nie działa!
c:\code> py 2 dualpkg\m2.py # Nie działa!

```

I odwrotnie, prosta instrukcja importu działa w trybie skryptu zarówno w wersji 2.x, jak i 3.x, a nie działa w trybie pakietu tylko w wersji 3.x, ponieważ w tej wersji takie instrukcje nie przeszukują katalogów pakietów:

```

# code\dualpkg\m2.py
import m1

c:\code> py 3
>>> import dualpkg.m2 # Nie działa!
c:\code> py 2
>>> import dualpkg.m2 # OK

c:\code> py 3 dualpkg\m2.py # OK
c:\code> py 2 dualpkg\m2.py # OK

```

I wreszcie użycie pełnych ścieżek pakietów działa w obu trybach użytkownika, jak i w obu wersjach Pythona, o ile katalog główny pakietu znajduje się w ścieżce wyszukiwania modułów (musi być tam dodany, aby mógł być używany gdzie indziej):

```

# code\dualpkg\m2.py
import dualpkg.m1 as m1 # oraz: set PYTHONPATH=c:\code

c:\code> py 3
>>> import dualpkg.m2 # OK
C:\code> py 2
>>> import dualpkg.m2 # OK

c:\code> py 3 dualpkg\m2.py # OK
c:\code> py 2 dualpkg\m2.py # OK

```

Podsumowując, o ile nie chcesz i nie możesz izolować modułów w podkatalogach poniżej katalogu skryptów, importowanie pełnych ścieżek pakietów jest prawdopodobnie lepszym rozwiązaniem niż importowanie względne wewnątrz pakietów — chociaż użycie pełnych ścieżek wymaga wpisania większej ilości kodu, jest rozwiązaniem bardziej uniwersalnym, obsługuje wszystkie przypadki zastosowania i działa tak samo zarówno w wersji 2.x, jak i 3.x. Oczywiście istnieją jeszcze inne sztuczki pozwalające na wykonanie takiego zadania, obejmujące dodatkowe czynności (np. ręczne ustawienie ścieżki `sys.path` w kodzie), ale pominiemy je tutaj, ponieważ są one bardziej zagmatwane i opierają się na wykorzystaniu semantyki importu, która jest podatna na błędy, podczas gdy importowanie z użyciem pełnych ścieżek opiera się tylko na podstawowych mechanizmach pakietów.

Oczywiście zakres, w jakim może to wpływać na Twoje moduły, może się różnić w zależności od pakietu; importy bezwzględne mogą wymagać wprowadzenia zmian do organizacji katalogów pakietów, a z kolei importy względne mogą się zakończyć niepowodzeniem, jeżeli moduł lokalny zostanie przeniesiony w inne miejsce.



Pamiętaj również o zmianach, jakie w tym zakresie mogą pojawić się w przyszłych wersjach języka Python. Chociaż ta książka dotyczy tylko Pythona do wersji 3.3, w dokumentach PEP (ang. *Python Enhancement Proposals*) mówi się o możliwościach rozwiązania niektórych problemów z pakietami w Pythonie 3.4, być może nawet zezwalając na względne importowanie w trybie skryptu. Z drugiej strony zakres i wynik tej inicjatywy jest niepewny i działałoby tylko w wersji 3.4 i nowszych; podane tutaj rozwiązanie z pełnymi ścieżkami jest uniwersalne i neutralne dla wersji. Oznacza to, że możesz czekać na zmiany wprowadzane w kolejnych wersjach linii 3.x lub po prostu używać sprawdzonych i zawsze działających pełnych ścieżek pakietów.

Pakiety przestrzeni nazw w Pythonie 3.3

Teraz gdy dowiedziałeś się już wielu rzeczy o pakietach i importowaniu względnym wewnątrz pakietów, warto również zauważyć, że istnieje nowa opcja, która modyfikuje niektóre omawiane tutaj wcześniej pomysły. Przynajmniej abstrakcyjnie, od wersji 3.3, Python ma cztery modele importu. Od najstarszego do najnowszego wyglądają one następująco:

Podstawowe, proste importowanie modułów: `import mod, from mod import attr`

Pierwszy, oryginalny model importowania: import plików i ich zawartości względem ścieżki wyszukiwania modułów `sys.path`.

Import pakietów: `import dir1.dir2.mod, from dir1.mod import attr`

Import, który dodaje rozszerzenia ścieżki katalogu względem ścieżki wyszukiwania modułu `sys.path`, gdzie każdy pakiet jest zawarty w jednym katalogu i ma plik inicjujący; dostępne w Pythonie 2.x i 3.x.

Import względem pakietu: `from . import mod (względny), import mod (bezwzględny)`

Model zastosowany do importu wewnątrz pakietów, omawianego w poprzedniej sekcji, wraz z jego względnymi lub bezwzględnymi schematami wyszukiwania dla importów z użyciem kropek lub `bez`; dostępne, ale różniący się w wersjach 2.x i 3.x Pythona.

Pakiety przestrzeni nazw: `import splitdir.mod`

Nowy model pakietu przestrzeni nazw, który omówimy w tym podrozdziale; pozwala pakietom składać się z wielu katalogów i nie wymaga pliku inicjującego; wprowadzony w Pythonie 3.3.

Pierwsze dwa modele są całkowicie samowystarczalne, trzeci zaostrza kolejność wyszukiwania i rozszerza składnię dla importów wewnątrz pakietu, a czwarty całkowicie odwraca niektóre podstawowe koncepcje i wymagania poprzedniego modelu pakietu. W praktyce spowodowało to, że Python 3.3 i nowsze wersje posiadają teraz tylko dwa warianty pakietów:

- Oryginalny model, obecnie znany jako *pakiety regularne* lub po prostu *zwykłe* (ang. *regular packages*).
- Model alternatywny, znany jako *pakiety przestrzeni nazw* (ang. *namespace packages*).

Jest to podobne w założeniu do dychotomii klasycznego i nowego stylu modeli klas, które spotkamy w następnej części tej książki, choć w przypadku pakietów nowy styl jest raczej dodatkiem do starego. Oryginalne i nowe modele pakietów nie wykluczają się wzajemnie i mogą być używane jednocześnie w tym samym programie. W rzeczywistości nowy model pakietów przestrzeni nazw działa jako *opcja rezerwowa*, używana tylko wtedy, gdy normalne moduły i standardowe pakiety o tej samej nazwie nie są obecne w ścieżce wyszukiwania modułów.

Uzasadnienie dla istnienia pakietów przestrzeni nazw jest zakorzenione w celach *instalacji* pakietów, które mogą wydawać się nie do końca oczywiste; jeżeli jesteś odpowiedzialny za takie zadania, powinieneś uważnie zapoznać się z dokumentem PEP dotyczącym tej funkcji. Krótko mówiąc, pakiety przestrzeni nazw radykalnie rozwiązują problem związany z możliwością wystąpienia kolizji wielu plików `__init__.py` podczas scalania części pakietu poprzez całkowite usunięcie tego pliku. Ponadto dzięki zapewnieniu standardowej obsługi pakietów, które można podzielić na wiele katalogów i umieszczać w kolejnych wpisach w ścieżce `sys.path`, pakiety przestrzeni nazw zwiększają elastyczność instalacji i zapewniają zintegrowany mechanizm zastępujący wiele niekompatybilnych ze sobą rozwiązań, które były używane do tej pory.

Chociaż jest jeszcze zbyt wcześnie, aby ocenić rzeczywistą przydatność pakietów przestrzeni nazw, przeciętni użytkownicy Pythona zwykle określają je jako bardzo użyteczne i alternatywne rozszerzenie zwykłego modelu pakietu — takie, które nie wymaga plików inicjujących i pozwala na użycie dowolnego katalogu kodu jako pakietu do zaimportowania. Aby przekonać się dla czego, przejdziemy do szczegółów.

Semantyka pakietów przestrzeni nazw

Pakiet przestrzeni nazw nie różni się zasadniczo od zwykłego pakietu; to po prostu nieco inny sposób tworzenia pakietów. Co więcej, na najwyższym poziomie nadal są one względne w stosunku do ścieżki `sys.path`: pierwszy komponent ścieżki pakietu przestrzeni nazw (zapisywanej z kropkami) musi nadal znajdować się w normalnej ścieżce wyszukiwania modułów.

Jednak pod względem budowy fizycznej oba rodzaje pakietów mogą się znacznie od siebie różnić. Zwykle pakiety nadal muszą być zapisane w jednym katalogu i posiadać plik `__init__.py`, który jest uruchamiany automatycznie. W przeciwieństwie do nich nowe pakiety przestrzeni nazw nie mogą zawierać pliku `__init__.py` i mogą składać się z wielu katalogów wykorzystywanych podczas importu. W rzeczywistości żaden z katalogów tworzących pakiet przestrzeni nazw nie może mieć pliku `__init__.py`, ale zawartość zagnieżdżona w każdym z nich jest traktowana jak pojedynczy pakiet.

Algorytm importu

Aby naprawdę zrozumieć pakiety przestrzeni nazw, musimy zajrzeć pod „maskę”, aby zobaczyć, jak działa operacja importowania w wersji 3.3. Podczas importu Python nadal iteruje po każdym katalogu w ścieżce wyszukiwania modułów — zdefiniowanej przez `sys.path` dla importów bezwzględnych oraz według lokalizacji pakietu dla importów względnych i komponentów zagnieżdżonych w ścieżkach pakietów — tak jak to się odbywało w Pythonie 3.2 i wersjach wcześniejszych. Jednak szukając zaimportowanego modułu lub pakietu o nazwie np. `spam` w wersji 3.3, dla każdego katalogu w ścieżce wyszukiwania modułów Python sprawdza większą liczbę kryteriów w następującej kolejności:

1. Jeżeli zostanie znaleziony plik `nazwa_katalogu\spam__init__.py`, importowany i zwracany jest zwykły pakiet.
2. Jeżeli zostanie znaleziony plik `nazwa_katalogu\spam.py` (lub `spam.pyc` lub inne rozszerzenie modułu), importowany i zwracany jest zwykły pakiet.
3. Jeżeli zostanie znaleziony katalog `nazwa_katalogu\spam`, zostanie on zapisany, a skanowanie będzie kontynuowane w następnym katalogu ze ścieżki wyszukiwania.
4. Jeżeli żaden z powyższych plików lub katalogów nie zostanie znaleziony, skanowanie będzie kontynuowane od następnego katalogu ze ścieżki wyszukiwania.

Jeżeli skanowanie ścieżki wyszukiwania zakończy się bez zwracania modułu lub pakietu według kroków 1. lub 2., a co najmniej jeden katalog został zapisany w kroku 3., wówczas tworzony jest *pakiet przestrzeni nazw*.

Tworzenie pakietu przestrzeni nazw odbywa się natychmiast i nie jest odraczane do momentu wystąpienia importu na poziomie niżej. Nowy pakiet przestrzeni nazw ma atrybut `__path__` ustawiony na iterowalną listę ścieżek katalogów, które zostały znalezione i zarejestrowane podczas skanowania w kroku 3., ale nie posiada atrybutu `__file__`.

Atrybut `__path__` jest następnie wykorzystywany w późniejszych, głębszych dostęпах do przeszukiwania wszystkich składników pakietu — każdy katalog w ścieżce `__path__` pakietu przestrzeni nazw jest przeszukiwany, gdy poszukiwane są bardziej zagnieżdżone elementy, podobnie jak to miało miejsce w przypadku katalogu zwykłego pakietu.

Patrząc z innej strony, atrybut `__path__` pakietu przestrzeni nazw pełni tę samą rolę dla komponentów niższego poziomu, co `sys.path` dla elementów ścieżek importu pakietów; staje się „ścieżką nadrzędną” pozwalającą na dostęp do położonych niżej elementów przy użyciu tej samej czteroetapowej procedury, którą omówiliśmy przed chwilą.

W efekcie pakiet przestrzeni nazw jest rodzajem *wirtualnej konkatencji* katalogów zdefiniowanych za pomocą wpisów w ścieżce wyszukiwania modułów. Jednak po utworzeniu pakietu przestrzeni nazw nie ma funkcjonalnej różnicy między nim a zwykłym pakietem; obsługuje wszystko, czego nauczyliśmy się dla zwykłych pakietów, w tym importowanie względne wewnątrz pakietu.

Wpływ na zwykłe pakiety: opcjonalne pliki `__init__.py`

Jedną z konsekwencji tej nowej procedury importowania jest to, że od wersji 3.3 Pythona pakiety nie muszą już posiadać plików `__init__.py` — gdy pakiet z jednym katalogiem nie ma tego pliku, będzie traktowany jako pakiet przestrzeni nazw z jednym katalogiem i żadne ostrzeżenie nie będzie wyświetlane. Jest to znaczne złagodzenie wcześniejszych zasad, ale zgodne z powszechnymi uwagami użytkowników; wiele pakietów nie wymaga kodu inicjalizacji, a mimo to w takich przypadkach i tak konieczne było utworzenie pustego pliku inicjalizacji — począwszy od wersji 3.3, nie jest to już wymagane.

Jednocześnie oryginalny, standardowy model pakietu jest nadal w pełni obsługiwany i automatycznie uruchamia kod znajdujący się w pliku `__init__.py`, co spełnia rolę *punktu zaczepienia dla inicjalizacji pakietu*. Ponadto gdy wiadomo, że pakiet nigdy nie będzie częścią podzielonego pakietu przestrzeni nazw, kodowanie go jako zwykłego pakietu z użyciem pliku `__init__.py` daje pewną *przewagę wydajności*. Tworzenie i ładowanie zwykłego pakietu następuje natychmiast, gdy zostanie on zlokalizowany w ścieżce wyszukiwania. W przypadku pakietów przestrzeni nazw przed utworzeniem pakietu muszą zostać przeskanowane wszystkie wpisy ze ścieżki wyszukiwania. Bardziej formalnie mówiąc, zwykłe pakiety zatrzymują algorytm z poprzedniej sekcji już na pierwszym kroku, podczas gdy pakiety przestrzeni nazw nie.

Zgodnie z dokumentem PEP opisującym tę zmianę, usuwanie obsługi zwykłych pakietów nie jest planowane — a przynajmniej tak to wygląda na dzień dzisiejszy; w projektach typu open source zmiany są zawsze możliwe (w poprzedniej edycji tej książki wspominaliśmy o planach zmian w metodach formatowania łańcuchów i względnego importu w wersji 2.x, które zostały później porzucone), więc jak zwykle powinniśmy uważnie śledzić przyszłe zmiany w tym

zakresie. Biorąc pod uwagę przewagę wydajności i możliwość automatycznej inicjalizacji zwykłych pakietów, wydaje się jednak mało prawdopodobne, że ich obsługa zostanie całkowicie usunięta.

Pakiety przestrzeni nazw w akcji

Aby zobaczyć, jak działają pakiety przestrzeni nazw w praktyce, przyjrzyjmy się dwóm następującym modułom i niezbędnej strukturze katalogów — z dwoma podkatalogami o nazwie *sub* zlokalizowanymi w różnych katalogach macierzystych, *dir1* i *dir2*:

```
C:\code\ns\dir1\sub\mod1.py
C:\code\ns\dir2\sub\mod2.py
```

Jeżeli dodamy zarówno *dir1*, jak i *dir2* do ścieżki wyszukiwania modułów, katalog *sub* staje się pakietem przestrzeni nazw obejmującym oba katalogi, z dwoma plikami modułów dostępnymi pod tą nazwą, pomimo że znajdują się one w osobnych katalogach fizycznych. Oto zawartość plików i wymagane ustawienia ścieżek w systemie Windows: nie ma tutaj plików `__init__.py` — w rzeczywistości *nie może ich być* w pakietach przestrzeni nazw, ponieważ jest to ich najważniejsza różnica fizyczna:

```
c:\code> mkdir ns\dir1\sub                # Dwa podkatalogi o tej samej nazwie w różnych katalogach
c:\code> mkdir ns\dir2\sub                # Podobnie w systemach innych niż Windows

c:\code> type ns\dir1\sub\mod1.py        # Pliki modułów w różnych katalogach
print(r'dir1\sub\mod1')

c:\code> type ns\dir2\sub\mod2.py
print(r'dir2\sub\mod2')
```

```
c:\code> set PYTHONPATH=C:\code\ns\dir1;C:\code\ns\dir2
```

Po bezpośrednim `zimportowaniu` w wersji 3.3 Pythona i nowszych pakiet przestrzeni nazw staje się *wirtualnym połączeniem* poszczególnych składników katalogu i umożliwia poprzez normalny import dostęp do dalszych zagnieżdżonych części za pomocą jednej, złożonej nazwy:

```
c:\code> C:\Python33\python
>>> import sub
>>> sub                                     # Pakiety przestrzeni nazw: zagnieżdżone ścieżki wyszukiwania
<module 'sub' (namespace)>
>>> sub.__path__
_NamespacePath(['C:\\code\\ns\\dir1\\sub', 'C:\\code\\ns\\dir2\\sub'])

>>> from sub import mod1
dir1\sub\mod1
>>> import sub.mod2                         # Zawartość z dwóch różnych katalogów
dir2\sub\mod2

>>> mod1
<module 'sub.mod1' from 'C:\\code\\ns\\dir1\\sub\\mod1.py'>
>>> sub.mod2
<module 'sub.mod2' from 'C:\\code\\ns\\dir2\\sub\\mod2.py'>
```

Dzieje się tak również wtedy, gdy *natychmiast* importujemy nazwę pakietu za pośrednictwem przestrzeni nazw — ponieważ pakiet przestrzeni nazw jest tworzony przy pierwszym użyciu, czas przeszukiwania ścieżek jest nieistotny:

```
c:\code> C:\Python33\python
>>> import sub.mod1
dir1\sub\mod1
```

```

>>> import sub.mod2 # Jeden pakiet składający się z dwóch katalogów
dir2\sub\mod2

>>> sub.mod1
<module 'sub.mod1' from 'C:\\code\\ns\\dir1\\sub\\mod1.py'>
>>> sub.mod2
<module 'sub.mod2' from 'C:\\code\\ns\\dir2\\sub\\mod2.py'>

>>> sub
<module 'sub' (namespace)>
>>> sub.__path__
_NamespacePath(['C:\\code\\ns\\dir1\\sub', 'C:\\code\\ns\\dir2\\sub'])

```

Co ciekawe, *import względny* działa również w pakietach przestrzeni nazw — pokazana poniżej instrukcja importu względnego odwołuje się do pliku w pakiecie, mimo że sam plik, do którego się odwołuje, znajduje się w *innym katalogu*:

```

c:\code> type ns\dir1\sub\mod1.py
from . import mod2 # próba wykonania "from . import string" nadal kończy się niepowodzeniem
print(r'dir1\sub\mod1')

c:\code> C:\Python33\python
>>> import sub.mod1 # Względny import modułu mod2 z innego katalogu
dir2\sub\mod2
dir1\sub\mod1
>>> import sub.mod2 # Wcześniej zaimportowany moduł nie jest wykonywany ponownie
>>> sub.mod2
<module 'sub.mod2' from 'C:\\code\\ns\\dir2\\sub\\mod2.py'>

```

Jak widać, pakiety przestrzeni nazw są pod każdym względem takie jak zwykłe pakiety z jednym katalogiem, z wyjątkiem dzielonego magazynu fizycznego — właśnie dlatego pakiety przestrzeni nazw składające się z *jednego katalogu* bez pliku `__init__.py` są dokładnie takie same jak zwykłe pakiety, z tym że nie posiadają kodu inicjującego, który mógłby być uruchamiany automatycznie.

Zagnieżdżanie pakietów przestrzeni nazw

Pakiety przestrzeni nazw obsługują nawet dowolne *zagnieżdżanie* — po utworzeniu pakietu przestrzeni nazw spełnia na swoim poziomie zasadniczo tę samą rolę, co ścieżka `sys.path` na górze, stając się „ścieżką nadrzędną” dla niższych poziomów. Kontynuując przykład z poprzedniej sekcji:

```

c:\code> mkdir ns\dir2\sub\lower # Głębiej zagnieżdżone komponenty
c:\code> type ns\dir2\sub\lower\mod3.py
print(r'dir2\sub\lower\mod3')

c:\code> C:\Python33\python
>>> import sub.lower.mod3 # Pakiet przestrzeni nazw zagnieżdżony w pakiecie przestrzeni nazw
dir2\sub\lower\mod3

c:\code> C:\Python33\python
>>> import sub # Ten sam rezultat przy ręcznym przechodzeniu do kolejnych poziomów
>>> import sub.mod2
dir2\sub\mod2

>>> import sub.lower.mod3
dir2\sub\lower\mod3

>>> sub.lower # Jednokatalogowy pakiet przestrzeni nazw
<module 'sub.lower' (namespace)>
>>> sub.lower.__path__
_NamespacePath(['C:\\code\\ns\\dir2\\sub\\lower'])

```

W powyższym przypadku `sub` jest pakietem przestrzeni nazw podzielonym na dwa katalogi, a `sub.lower` jest jednokatalogowym pakietem przestrzeni nazw zagnieżdżonym w części `sub` fizycznie zlokalizowanej w katalogu `dir2`. `sub.lower` jest także pakietem przestrzeni nazw będącym odpowiednikiem zwykłego pakietu bez pliku `__init__.py`.

Takie zachowanie zagnieżdżania jest prawdziwe niezależnie od tego, czy niższy komponent jest modulem, zwykłym pakietem, czy innym pakietem przestrzeni nazw — jako nowe ścieżki wyszukiwania importów pakiety przestrzeni nazw pozwalają na swobodne zagnieżdżenie w nich wszystkich trzech rodzajów komponentów:

```
c:\code> mkdir ns\dir1\sub\pkg
C:\code> type ns\dir1\sub\pkg\__init__.py
print(r'dir1\sub\pkg\__init__.py')

c:\code> C:\Python33\python
>>> import sub.mod2 # Zagnieżdżony moduł
dir2\sub\mod2
>>> import sub.pkg # Zagnieżdżony zwykły pakiet
dir1\sub\pkg\__init__.py
>>> import sub.lower.mod3 # Zagnieżdżony pakiet przestrzeni nazw
dir2\sub\lower\mod3

>>> sub # Moduły, pakiety zwykłe i pakiety przestrzeni nazw
<module 'sub' (namespace)>
>>> sub.mod2
<module 'sub.mod2' from 'C:\\code\\ns\\dir2\\sub\\mod2.py'>
>>> sub.pkg
<module 'sub.pkg' from 'C:\\code\\ns\\dir1\\sub\\pkg\\__init__.py'>
>>> sub.lower
<module 'sub.lower' (namespace)>
>>> sub.lower.mod3
<module 'sub.lower.mod3' from 'C:\\code\\ns\\dir2\\sub\\lower\\mod3.py'>
```

Aby jeszcze lepiej zrozumieć działanie pakietów, powinieneś uważnie przeanalizować pliki i katalogi z powyższego przykładu. Jak widać, pakiety przestrzeni nazw bezproblemowo integrują się z poprzednimi modelami importów i rozszerzają je o nowe funkcje.

Pliki nadal mają pierwszeństwo przed katalogami

Jak wyjaśnialiśmy już wcześniej, jednym z zadań plików `__init__.py` w zwykłych pakietach jest zadeklarowanie katalogu jako pakietu — taki plik mówi Pythonowi, aby używał całego katalogu, zamiast poszukiwać możliwego pliku o tej samej nazwie w dalszej ścieżce. Pozwala to uniknąć przypadkowego wybrania podkatalogu niezawierającego żadnego kodu, który przypadkowo pojawiłby się gdzieś w ścieżce wyszukiwania modułów przed właściwym modulem o tej samej nazwie.

Ponieważ pakiety przestrzeni nazw nie wymagają tych specjalnych plików, może się wydawać, że neutralizują to zabezpieczenie. Tak jednak nie jest — ponieważ opisany wcześniej algorytm przestrzeni nazw kontynuuje skanowanie ścieżki wyszukiwania po znalezieniu katalogu przestrzeni nazw, pliki znalezione później na tej ścieżce nadal mają wyższy priorytet niż wcześniejsze katalogi bez pliku `__init__.py`. Weźmy na przykład następujące katalogi i moduły:

```
c:\code> mkdir ns2
c:\code> mkdir ns3
c:\code> mkdir ns3\dir
c:\code> notepad ns3\dir\ns2.py
c:\code> type ns3\dir\ns2.py
print(r'ns3\dir\ns2.py!')
```

Katalogu `ns2`, pokazanego tutaj, nie można zaimportować w Pythonie 3.2 i wersjach wcześniejszych — nie jest to zwykły pakiet, ponieważ nie ma w nim pliku inicjującego `__init__.py`. Katalog ten można jednak zaimportować w wersji 3.3 — jest to katalog pakietu przestrzeni nazw w bieżącym katalogu roboczym, który zawsze jest *pierwszym* elementem ścieżki wyszukiwania modułów `sys.path`, niezależnie od ustawień zmiennej `PYTHONPATH`:

```
c:\code> set PYTHONPATH=
c:\code> py 3.2
>>> import ns2
ImportError: No module named ns2

c:\code> py 3.3
>>> import ns2
>>> ns2                                     # Jednokatalogowy moduł przestrzeni nazw w bieżącym katalogu roboczym
<module 'ns2' (namespace)>
>>> ns2.__path__
_NamespacePath(['.\ns2'])
```

Ale zobacz, co się stanie, gdy katalog zawierający plik o tej samej nazwie co katalog przestrzeni nazw zostanie dodany *później* na ścieżce wyszukiwania za pomocą ustawień zmiennej `PYTHONPATH` — zamiast katalogu zostanie użyty plik, ponieważ Python kontynuuje przeszukiwanie kolejnych pozycji ścieżki wyszukiwania modułów po znalezieniu katalogu pakietu przestrzeni nazw. Proces ten jest przerywany dopiero wtedy, gdy znaleziony zostaje pasujący moduł lub zwykły pakiet albo ścieżka wyszukiwania zostaje całkowicie przeskanowana do końca. Pakiety przestrzeni nazw są zwracane tylko wtedy, gdy po drodze nie znaleziono niczego innego:

```
c:\code> set PYTHONPATH=C:\code\ns3\dir
c:\code> py 3.3
>>> import ns2                               # Użyj późniejszego pliku modułu, a nie katalogu o tej samej nazwie!
ns3\dir\ns2.py!
>>> ns2
<module 'ns2' from 'C:\code\ns3\dir\ns2.py'>

>>> import sys
>>> sys.path[:2]                             # Pierwszy element '' reprezentuje bieżący katalog roboczy
['', 'C:\code\ns3\dir']
```

W rzeczywistości ustawienie ścieżki w celu dołączenia modułu działa tak samo jak we wcześniejszych wersjach Pythona, nawet jeżeli katalog przestrzeni nazw o tej samej nazwie pojawia się wcześniej na ścieżce; pakiety przestrzeni nazw są używane w wersji 3.3 tylko w przypadkach, które zakończyłyby się wystąpieniem błędów we wcześniejszych wersjach Pythona:

```
c:\code> py 3.2
>>> import ns2
ns3\dir\ns2.py!
>>> ns2
<module 'ns2' from 'C:\code\ns3\dir\ns2.py'>
```

Z tego też powodu *żadne* katalogi w pakiecie przestrzeni nazw nie mogą zawierać pliku `__init__.py`: gdy tylko algorytm importu znajdzie plik o takiej nazwie, natychmiast zwraca pakiet standardowy i kończy przeszukiwanie ścieżki i przestrzeni nazw. Mówiąc bardziej formalnie, algorytm importu wybiera pakiet przestrzeni nazw tylko na końcu skanowania ścieżki wyszukiwania i zatrzymuje się na krokach 1. lub 2., jeżeli wcześniej zostanie znaleziony zwykły plik pakietu lub moduł.

W rezultacie *zarówno* pliki modułów, jak i zwykle pakiety w dowolnym miejscu ścieżki wyszukiwania modułów mają pierwszeństwo przed katalogami pakietów przestrzeni nazw. W poniższym przykładzie pakiet przestrzeni nazw o nazwie `sub` istnieje jako konkatenacja podkatalogów o takich samych nazwach z katalogu `dir1` i `dir2` na ścieżce wyszukiwania:

```
c:\code> mkdir ns4\dir1\sub
c:\code> mkdir ns4\dir2\sub
c:\code> set PYTHONPATH=c:\code\ns4\dir1;c:\code\ns4\dir2
c:\code> py 3
>>> import sub
>>> sub
<module 'sub' (namespace)>
>>> sub.__path__
_NamespacePath(['c:\code\ns4\dir1\sub', 'c:\code\ns4\dir2\sub'])
```

Jednak podobnie jak plik modułu, tak i *zwykły pakiet* dodany na końcu ścieżki wyszukiwania ma pierwszeństwo przed katalogami pakietów przestrzeni nazw o takiej samej nazwie — skanowanie ścieżki wyszukiwania zaczyna wstępnie zapisywać pakiet przestrzeni nazw w katalogu `dir1` jak poprzednio, ale porzuca go, gdy zwykły pakiet zostanie wykryty w katalogu `dir2`:

```
c:\code> notepad ns4\dir2\sub\__init__.py
c:\code> py 3
>>> import sub                                     # Użyj zwykłego pakietu z końca ścieżki, a nie katalogu o tej samej
nazwie
>>> sub
<module 'sub' from 'c:\code\ns4\dir2\sub\__init__.py'>
```

Choć jest to przydatne rozszerzenie, to jednak ze względu na fakt, że pakiety przestrzeni nazw są dostępne tylko dla użytkowników wersji 3.3 Pythona (i nowszych), nie będziemy go tutaj bardziej szczegółowo omawiać; więcej szczegółowych informacji na ten temat znajdziesz w dokumentacji Twojej wersji Pythona. Jeżeli jesteś zainteresowany zagadnieniami związanymi z importowaniem, powinieneś uważnie zapoznać się zwłaszcza z dokumentem PEP tej zmiany, który zawiera szczegółowe uzasadnienie, dodatkowe informacje i bardziej wyczerpujące przykłady zastosowania.

Podsumowanie rozdziału

W niniejszym rozdziale wprowadziliśmy *model importowania pakietów* Pythona — opcjonalną, lecz przydatną metodę jawnego wymienienia części ścieżki katalogów prowadzących do modułów. Importowanie pakietów nadal odbywa się względem katalogu ze ścieżki wyszukiwania modułów, jednak zamiast pozostawić Pythonowi ręczne przejście ścieżki wyszukiwania, skrypt może podać resztę ścieżki do modułu w sposób jednoznaczny.

Jak widzieliśmy, pakiety nie tylko sprawiają, że importowanie w większych systemach jest bardziej zrozumiałe, ale także upraszczają ustawienia ścieżki wyszukiwania (jeżeli wszystkie operacje importowania pomiędzy katalogami odbywają się względem wspólnego katalogu głównego). Pomagają one również usunąć niejednoznaczność w sytuacji, gdy istnieje większa liczba modułów o tej samej nazwie (dodanie nazwy katalogu zawierającego moduł do importu pakietu pomaga odróżnić moduły).

Omówiliśmy również *model importów względnych*, stosowany wyłącznie w pakietach. Jest to sposób wymuszenia importu plików zdefiniowanych w tym samym pakiecie polegający na zastosowaniu wiodących kropek w ścieżce importu w instrukcji `from`. Metoda ta zastępuje stosowaną w starszych wersjach Pythona i bardziej podatną na błędy regułą domyślnego wy-

szukiwania modułów w bieżącym katalogu pakietu. Na koniec zbadaliśmy *pakiety przestrzeni nazw* Pythona 3.3, które pozwalają, aby pakiet składał się z wielu katalogów fizycznych; jest to w pewnym sensie „ostatnia deska ratunku” podczas wyszukiwania importów, która całkowicie usunęła wymagania dotyczące plików inicjalizacyjnych znane z poprzedniego modelu importowania.

W kolejnym rozdziale omówimy kilka bardziej zaawansowanych zagadnień związanych z modułami, takich jak użycie atrybutu `__name__` czy importowanie nazw. Jak zawsze jednak zamknijemy bieżący rozdział krótkim quizem sprawdzającym wiadomości w nim przedstawione.

Sprawdź swoją wiedzę — quiz

1. Jaki jest cel umieszczania pliku `__init__.py` w katalogu pakietu modułu?
2. W jaki sposób możemy uniknąć powtarzania pełnej ścieżki pakietu za każdym razem, gdy odnosimy się do zawartości pakietu?
3. Które katalogi wymagają, by znajdował się w nich plik `__init__.py`?
4. Kiedy w przypadku importowania pakietów musimy użyć instrukcji `import` zamiast `from`?
5. Jaka jest różnica między instrukcją `from mypkg import spam` a `from . import spam`?
6. Czym jest pakiet przestrzeni nazw?

Sprawdź swoją wiedzę — odpowiedzi

1. Plik `__init__.py` służy do deklarowania i inicjalizacji zwykłego pakietu modułu. Python automatycznie wykonuje jego kod za pierwszym razem, gdy w procesie importujemy moduł za pośrednictwem katalogu. Przypisane zmienne pliku stają się atrybutami obiektu modułu utworzonego w pamięci i odpowiadającego temu katalogowi. Nie jest on opcjonalny — nie możemy importować modułów za pomocą składni pakietów, jeżeli katalog nie zawiera tego pliku.
2. Aby bezpośrednio skopiować zmienne z pakietu, należy użyć instrukcji `from` lub rozszerzenia `as` w połączeniu z instrukcją `import`, zastępując ścieżkę krótszym synonimem. W obu przypadkach ścieżka wymieniana jest tylko w jednym miejscu, czyli instrukcji `from` bądź `import`.
3. W Pythonie 3.2 i wersjach wcześniejszych każdy katalog wymieniony w instrukcjach `import` oraz `from` musiał zawierać plik `__init__.py`. Pozostałe katalogi, w tym katalog zawierający pierwszy od lewej strony komponent ścieżki pakietu, nie muszą zawierać tego pliku.
4. W przypadku pakietów instrukcji `import` musimy użyć w miejsce `from` jedynie wtedy, gdy potrzebujemy uzyskać dostęp do tej samej zmiennej zdefiniowanej w więcej niż jednej ścieżce. Dzięki instrukcji `import` ścieżka sprawia, że referencje stają się unikalne, natomiast instrukcja `from` pozwala na wykorzystywanie tylko jednej wersji danej nazwy zmiennej (o ile do zmiany nazwy nie użyjesz rozszerzenia `as`).
5. W Pythonie 3.x instrukcja `from mypkg import spam` definiuje `import` *bezwzględny*: pakiet `mypkg` jest wyszukiwany z pominięciem katalogu pakietu, w którym występuje ta instrukcja, to znaczy przeszukiwana jest wyłącznie ścieżka `sys.path`. Instrukcja `from . import spam`

definiuje import *względny*: nazwa `spam` jest poszukiwana w odniesieniu do pakietu, w którym występuje ta instrukcja. W Pythonie 2.x import bezwzględny najpierw przeszukuje katalog pakietu, zanim przejdzie do ścieżki `sys.path`; import względny działa tak, jak to wcześniej opisywaliśmy.

6. Pakiet przestrzeni nazw jest rozszerzeniem modelu importu dostępnym w Pythonie 3.3 i nowszych wersjach; reprezentuje pakiet składający się z jednego lub większej liczby katalogów, które nie zawierają plików `__init__.py`. Gdy Python znajdzie takie katalogi podczas wyszukiwania importu i nie znajdzie wcześniej prostego modułu lub zwykłego pakietu o takiej nazwie, tworzy pakiet przestrzeni nazw, który jest wirtualnym połączeniem wszystkich znalezionych katalogów o nazwie odpowiadającej nazwie żądanego modułu. Dalsze zagnieżdżone komponenty są wyszukiwane we wszystkich katalogach pakietu przestrzeni nazw. W rezultacie powstaje pakiet bardzo podobny do zwykłego pakietu, ale jego zawartość może być podzielona na wiele katalogów.

PROGRAM PARTNERSKI

— GRUPY HELION —

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA
Helion

Już dziś zacznij pisać znakomity kod w Pythonie!

Python jest wieloparadygmatowym, wszechstronnym językiem programowania, zoptymalizowanym pod kątem efektywności pracy, czytelności kodu i jakości oprogramowania. Jego popularność rośnie, co wynika z wielości i różnorodności zastosowań oraz z tego, że jest darmowy i łatwo przenośny, można się go szybko nauczyć, a tworzenie kodu Pythona sprawia sporo przyjemności. Wszystkie te cechy dają zespołom deweloperskim strategiczną przewagę w dużych i małych projektach. Aby wykorzystać zalety Pythona, konieczne jest zdobycie solidnych podstaw tego języka, a następnie dogłębne zrozumienie bardziej zaawansowanych koncepcji i porządne ich przećwiczenie podczas pisania własnego kodu.

To kompleksowy podręcznik do nauki programowania w Pythonie. Jego piąte wydanie zostało gruntownie zaktualizowane i rozbudowane o dodatkowe treści. Omówiono tu obie wersje Pythona, czyli linie 3.X i 2.X, i dodano opisy nowych lub rozszerzonych mechanizmów, takich jak obsługa formatu JSON, moduł `timeit`, pakiet `PyPy`, metoda `os.popen`, generatory, rekurencje, słabe referencje, atrybuty i metody `__mro__`, `__iter__`, `super`, `__slots__`, metaklasy, deskryptory, funkcja `random`, pakiet `Sphinx` i wiele innych. W książce znalazło się mnóstwo ćwiczeń, quizów, pomocnych ilustracji oraz przykładów kodu. Jest to kompendium dla każdego, kto chce szybko zacząć programować w Pythonie i tworzyć wydajny kod o wysokiej jakości.

Mark Lutz — od 30 lat zajmuje się programowaniem. Dziś jest jedną z najważniejszych postaci w świecie Pythona. Napisał kilka popularnych, wielokrotnie wznawianych książek o programowaniu w tym języku. Przeprowadził też kilkaset sesji treningowych poświęconych Pythonowi. Zanim w 1992 roku zainteresował się tym językiem, zajmował się implementacją Prologa i pracował nad kompilatorami, narzędziami programistycznymi, aplikacjami skryptowymi oraz systemami klient-serwer.

W tej książce przedstawiono między innymi:

- składnię Pythona i koncepcje związane z iteracjami
- dokładny opis ważniejszych wbudowanych typów obiektów i ich możliwości
- struktury programistyczne wyższego poziomu
- programowanie funkcyjne i programowanie zorientowane obiektowo
- zaawansowane zagadnienia dla profesjonalistów

	<p>Sprawdź nasze szkolenia!</p>  <p>AKADEMIA IT & BUSINESS</p>	<p>KOD KORZYŚCI Sięgnij po więcej! ▶</p> 
 helion.pl	<p>HELIONSAZKOLENIA.PL</p>	<p>ISBN 978-83-283-6150-8</p>
<p>HELION SA ul. Kosciuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl</p>	<p>9 788328 361508</p>	<p>Cena: 169,00 zł</p>